2009-01-14

Application-Directed DVFS using Multiple Clock Domains on Graphics Hardware

Juan Li
Worcester Polytechnic Institute

Application-Directed DVFS using Multiple Clock Domains on Graphics Hardware

by Juan Li

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

Dec 2008

APPROVED:

_____ Professor Emmanuel O. Agu, Major Thesis Advisor

_____ Professor Robert E. Kinicki, Thesis Reader

_____ Professor Michael A. Gennert, Head of Department

Abstract

As handheld devices have become increasingly popular, powerful programmable graphics hardware for mobile and handheld devices has been deployed. While many resources on mobile devices are limited, the predominant problem for mobile devices is their limited battery power. Several techniques have been proposed to increase the energy efficiency of mobile applications and improve battery life.

In this thesis, we propose a new dynamic voltage and frequency scaling (DVFS) on Graphics Processing Units (GPU). In most cases, cues within the graphics application can be used to predict portions of a GPU that will be used or unused when the application is run. We partition the GPU into six clock domains that can be clocked at different rates. Specifically, each domain it has its own voltage and frequency setting based on its predicted workload to save energy without reducing applications frame rates. In addition, we propose an signature-based algorithm for predicting the workload offered to our six clock domains by a given application to decide voltage and frequency settings. We conduct experiments and compare the results of our new signature based workload prediction algorithm with some other traditional interval based workload prediction algorithms. Our results show that our signature-based prediction can save 30-50% energy without affecting application frame rates.

Acknowledgements

I would like to express my gratitude to my advisor Emmanuel O. Agu who gave me a lot of help for my research and study in recent two years. He gives me invaluable advice for the work and for life which can be remembered and be beneficial for my entire life.

Thanks also to the faculty in the cs department. They are kind and very supportive. The colleagues at the ISRG lab also gave me many wonderful moments.

My thanks go also to my reader Robert E. Kinicki who spent time for reading my thesis.

Thanks also to lots of friends, thanks for the strength given to me when I was frustrated. Thanks to my parents and my brother always who are there for me.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Researchers have recently become interested in using Graphics Processing Units (GPU) for graphics and non-graphics applications as GPUs now possess powerful computing ability and are fully programmable. In the last five years the computational ability of GPUs, measured as Floating Point Operations per Second (FLOPS), has increased five times, which significantly outpaces the often-quoted Moore's Law (2 x increases every 18 months for cpus). Figure 1.1 shows the rate measured by the number of float operations per second (FLOPS) at which these graphics cards have been improving. In addition to their performance improvements, recent graphics hardware architectures can be programmed to implement novel graphics (and non-graphic) algorithms. Essentially, these GPUs have moved away from traditional fixed-function pipelines to a new programmatically reconfigurable graphics pipeline.

GPUs on mobile devices are becoming common. More Personal Digital Assistants (PDAs) and smart phone cores now integrate programmable graphics hardware

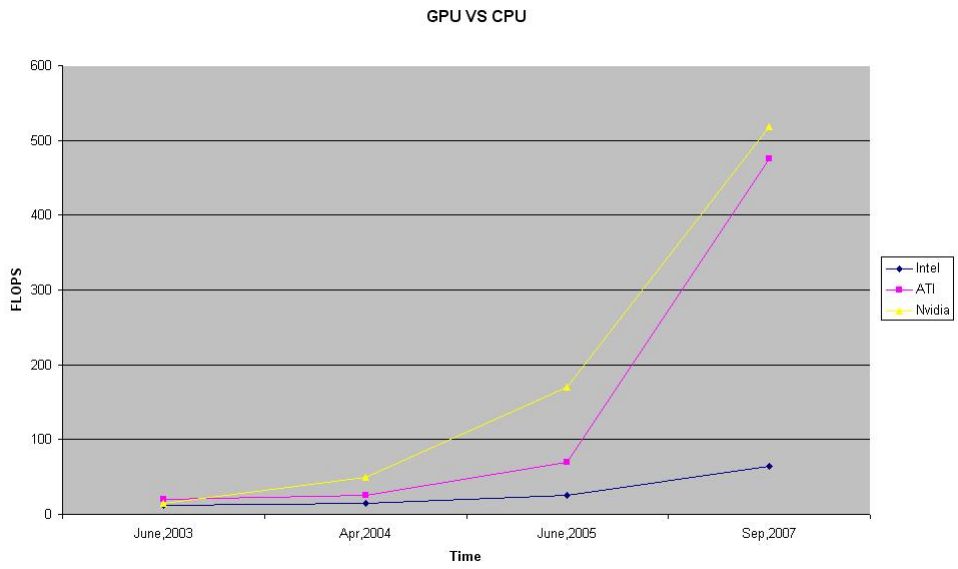

Figure 1.1: GPU vs. CPU Trends

for more realistic display. In fact, the new OpenGL ES standard has replaced the entire fixed function graphics pipeline with programmable shaders. However, almost half of the total power consumption of a mobile device is due to the display and graphics card [14]. Thus, techniques to save the battery energy of mobile devices while running graphics applications are especially important and interesting.

Indeed available battery energy has become the bottleneck since battery technology has been the slowest technology to advance in mobile devices. Figure 1.2 shows the technologies improvement in the last decade. Starner and Paradiso [33] showed that, from 1990 to 2003, battery energy density in mobile computing has only increased three-fold while other areas such as disk capacity, processor speed and available memory have increased over 250 times. Consequently, many researchers now focus on developing energy-efficient algorithms and software for mobile devices.

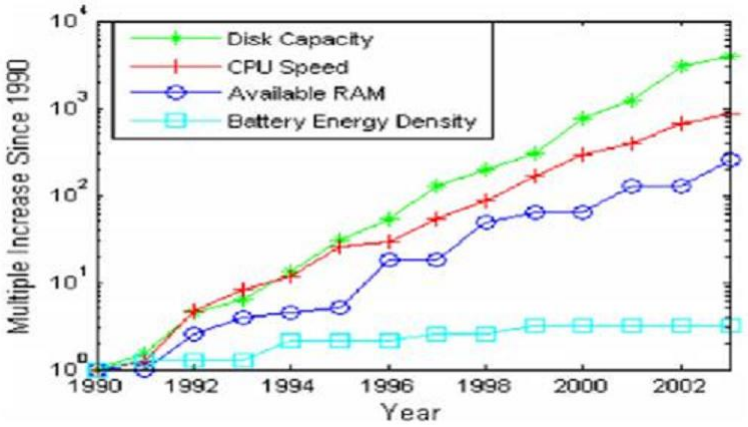

Figure 1.2: Advances in Computer and Battery Technologies([33])

1.2 Thesis Goal

In this work, we study the architecture of contemporary graphics hardware and the energy-saving techniques which are traditionally applied to CPU systems. We analyze the possibilities of applying dynamic voltage and frequency scaling system to a single graphics processor unit. Our main contributions in this work include:

1) Graphics architecture is defined with six clock domains according to their functions and uses .

2) The multiple clock domain architecture is implemented in a graphics hardware simulator and how multiple clock domains work to save energy is investigated.

3) A detailed quantitative workload characterization of the test graphics applications is provided. The usage for each domain is analyzed by the application and several statistics are collected. Such statistics can be used in future to guide the development of mobile device architecture.

4) Signature based algorithms is proposed to predict the workload offered to each of our clock domains .

5) It is argued that applications know best about their resource or energy needs

and the performance of API setting policy is compared with a simple polling algorithm.

6) Experimental studies are conducted to compare different prediction algorithms and it is illustrated that the proposed prediction algorithms save significant energy without noticeable performance change.

1.3 Thesis Organization

The rest of the thesis is as follows: In Chapter 2 we discuss background work about Dynamic Voltage and Frequency Scaling (DVFS) and describe Qsilver, a simulation framework for GPU and the general architecture of current consumer GPUs. In Chapter 3 we introduce our multiple clock domains and application level energy-efficient framework. Chapter 4 describes the prediction algorithms to decide the voltage and clock setting of the defined multiple clock domains for the incoming workloads. In chapter 5 we describe our experimental settings. Experimental results and discussions are analyzed in chapter 6. Chapter 7 introduces the related work and the conclusion and future work are presented in chapter 8.

Chapter 2

Background

In this chapter, the factor of battery energy is described and the technique of dynamic voltage and frequency scaling is introduced. After introducing energy and energy saving techniques, a graphics hardware simulator and the general graphics hardware architecture are reviewed.

2.1 Battery Energy

The energy E, measured in Joules (J), consumed by a computer over T seconds is equal to the integral of the instantaneous power, measured in Watts (W). The instantaneous power consumed by components implemented in CMOS, such as microprocessors and DRAM, is

$$E = CV^2F \tag{2.1}$$

where C is the capacitance, V is the voltage supplied to the component, and F is the frequency of the clock driving the component. Thus, the power consumed by a task may be reduced by reducing V , F or both. However, for tasks that require a fixed

amount of work, reducing the frequency may simply take more time to complete the work. Thus, little or no energy will be saved. Voltage or frequency reductions should thus be done intelligently.

There are some well-known techniques that can result in energy savings when the processor is idle, typically through clock gating, which avoids powering unused devices. A typical example is the power saving mode of the computer. When the system is idle for a period, the display or the hard disk can be set to idle to save the power.

However, as discussed by Linden et al. in [20], in practice, the amount of energy a battery can deliver (i.e., its capacity) is reduced with increased power consumption. When the system is idle, the processor core is disabled but the devices remain active. If the system clock is 206 MHz, a typical pair of alkaline batteries will power the system for about 2 hours; if the system clock is set to 59 MHz, those same batteries will last for about 18 hours. Although the battery lifetime increased by a factor of 9, in this case, the processor speed was only decreased by a factor of 3.5[21].

Another important characteristic for the capacity of the battery is that it can be increased by interspacing periods of high power demand with much longer periods of low power demand [7]. Those two characteristics are determined by the chemical properties and the construction of a battery as well as the conditions under which the battery is used. Therefore it is better to reduce the clock speed to the minimum needed rather than running at peak speed and then being idle. Grunwald [12] shows an example that normally takes 600 million instructions to complete. That application would take one second on a StrongARM SA-2 at 600MHz and would consume 500 mJoules. At 150MHz, the application would take four seconds to

complete, but would only consume 160 mJoules, a four-fold savings assuming that an idle computer consumes no energy. There is obviously a significant benefit to running slower when the application can tolerate additional delay.

There is numerous research on exploiting these two properties of the battery to maximize its capacity. Pering [36] used the term voltage scheduling to mean scheduling policies that seek to adjust both clock speeds and energy. The goal of voltage scheduling is to reduce the clock speed such that all work on the processor can be completed on time and then reducing the voltage to the minimum needed to insure stability at that frequency.

2.2 Dynamic Voltage and frequency Scaling(DVFS)

Dynamic voltage and frequency scaling (DVFS) is a technique widely used for reducing energy consumption of processors by varying the voltage and frequency at run time [16]. The main idea is to reduce frequency or voltage during periods when the processor has a reduced workload. If DVFS is done intelligently, energy can be saved without any perceptible effects on the speed at which the processor performs its tasks. Most systems are designed with fixed voltage and frequency settings in order to make the system stable. However, the activity levels of applications are variable, and in many cases, applications have idle periods when no useful task is performed. By reducing the processor voltage and frequency levels at run-time when the application has low-activity or idle periods, mobile battery energy can be saved without a noticeable impact on the performance.

DVFS has been applied to many CPU-based applications at both application levels and operating system levels. Some research by Liu et al [22] shows that 32-

50% energy savings can be achieved by various applications from multimedia to web browsing.

Recently, researchers have become interested in applying DVFS to GPUs. Jeabin et al [19] have demonstrated varying the clock frequency and voltage of traditional fixed-function graphics hardware based on different workload between the geometry processor and rendering engine. However, new GPU architectures use a programmable shader model. In this new framework, previously proposed DVFS techniques to DVFS based on balancing the workload between the geometry processor and rendering engine become obsolete. A new DVFS scheme suitable for new graphics hardware architecture needs to be proposed. Additionally, we would also like DVFS to be controlled by the application and triggered based on the workload predicted by examining the application.

2.3 Qsilver

Qsilver is a flexible simulation framework for graphics hardware architectures [39]. It simulates the architecture of a consumer graphics card Nvidia GeForce 4 while offering the possibilities of extending the simulated architecture to more advanced one.

Qsilver is a cycle accurate and queue driven system. There are four data queues to separate the functional units in the graphics hardware pipeline, which makes it possible to analyze the performance and detect the bottlenecks to each unit. A cycle-timer is used to count the number of operations and estimate the computational load or power cost. Qsilver takes the data stream that is obtained from Chromium, an OpenGL application wrapper, as the input that drives the data stream flowing through each unit inside the architecture. The time cost and energy

cost can be estimated from data collected and analyzed after the run finishes. The data stream input from Chromium is statistics data such as the numbers of pixels rendered, vertex, texels, lights and textures. These statistics data can be used for the estimation of the workload of each unit of the GPU.

Qsilver is a good candidate for research in the graphics hardware architecture. It simulates the functional units inside consumer hardware. It uses the trace and statistics data intercepted from a real openGL application that has been run at interactive speeds. It also offers interface to add new function units or change the interface between each unit.

However, Qsilver is designed for a fixed function pipeline and has limited computation power that was suitable for the hardware at that time (circa 2004). To use Qsilver for current GPU architectures or future graphics hardware architect, we need to extend Qsilver to support a programmable pipeline in graphics hardware. Additionally, the traditional fixed-function pipeline used in Qsilver is not amenable to current hardware architectures. We implement a programmable architecture based on the current Qsilver system. Also we shall discuss about the uni-shader architecture which is the new trend in the design of state-in-art graphics hardware.

2.4 Architecture of GPU

In the last four or five years, graphics hardware has shown more dramatic performance improvements than CPUs. It is not only the exponential increase in the computational power but also the programmable ability that facilitates more realistic rendering at more interactive speeds. Graphics hardware has evolved into a standardized architecture since they are primarily designed to run the standardized

rendering framework such as OpenGL or DirectX. Figure 2.1 shows the predominant architecture for current graphics hardware.

GPUs are designed as specific purpose processors applying sequences of operations to a stream of data. They take as input a stream of vertices that defines the geometry of the scene. The input vertex stream passes through a computation stage that transforms and computes some of the vertex attributes generating a stream of transformed vertices. The stream of transformed vertices is assembled into a stream of triangles, each triangle keeping the attributes of its three vertices. The stream of triangles may pass through a stage that performs a clipping test. Then each triangle passes through a rasterizer that generates a stream of fragments, discrete portions of the triangle surface that correspond with the pixels of the rendered images, on which fragment attributes are derived from the triangle vertex attributes.

This stream of fragments may pass through a number of stages performing a number of visibility tests (stencil, depth, alpha and scissor) that will remove non-visible fragments and then the stream of fragments will pass through a second computation stage. This second fragment computation stage may modify the fragment attributes using additional information from dimensional arrays stored in memory (textures). The stream of shaded fragments will finally update the render buffer and be displayed as pixels in the screen.

Modern GPUs implement the two described computation stages as programmable stages named vertex shading and fragment shading. The programmability of these stages and the streaming nature of the rendering algorithm allow the implementation of other stream-based algorithms over modern GPUs.

In addition, a "unified shader model" is now common where the distinction be-

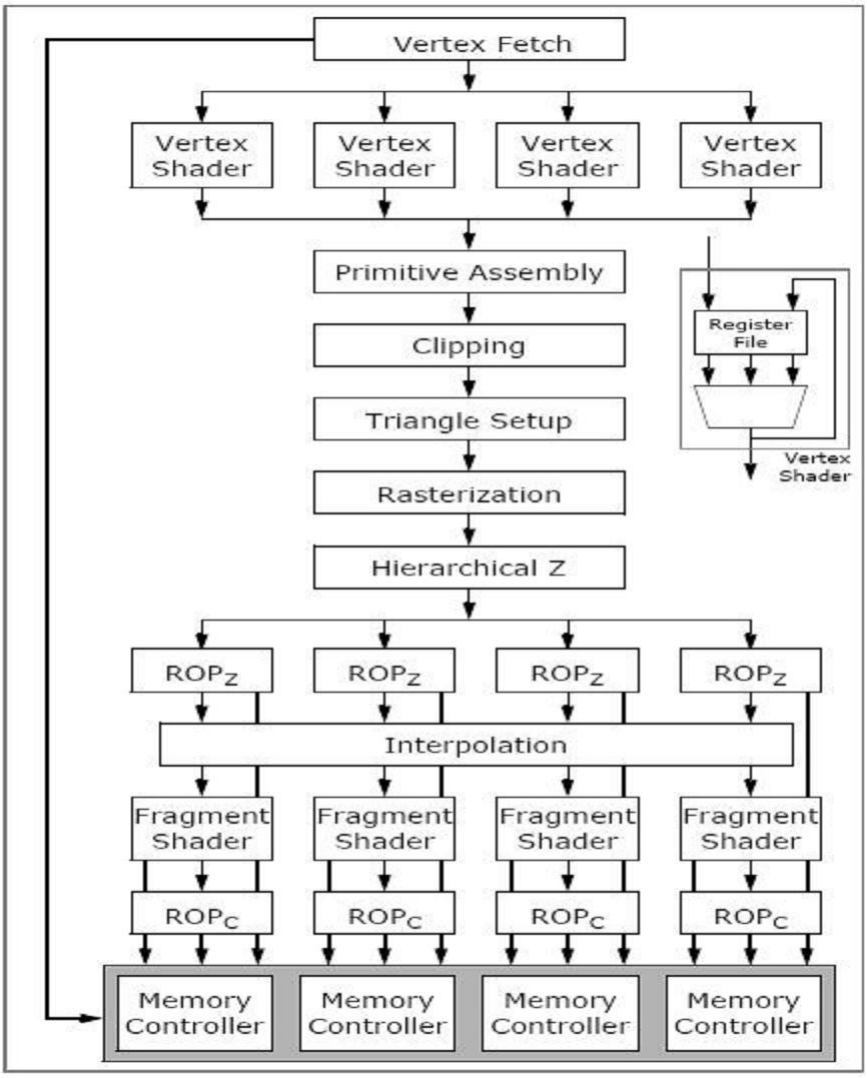

Figure 2.1: GPU Architecture ([26])

tween vertex shaders and pixel shaders is removed, providing a more generalized concept of shader units. Thus, the GPU can work on certain types of data, which wouldn't necessarily have to be defined as vertices or pixels. This offers developers ultimate flexibility and increased power. The apparent advantage of sharing pipelines is to add more assembly lines, making computation that much faster. Usually such hardware is composed of an array of computing units and a dynamic scheduling / load balancing unit that distributes shader work to the computing units. The NVidia Geforce 8 series which came out in 2007 are the first chipsets with the unified shader architecture in the world [32]. This architecture will be dominant in the consumer graphics hardware market and some previous technologies with old architecture need to be updated. The shader processor is the core unit to execute shaders. Instead of the traditional separated vertex and fragment processors, the latest graphics architect adopts a unified shader model which can execute the vertex shader, geometry shader, fragment shader (and maybe a new computing shader) with the same units. Apart from the difference that shaders are becoming smaller and smaller, graphics hardware vendors can use the unified interface to decrease the cost and increase the efficiency. In modern 3D graphics applications, shaders are the most important part to bring realistic and interactive scenes to users. More sophisticate models and lighting models can be implemented to create a realistic effect.

Chapter 3

Application Level Energy-efficient Framework

3.1 System Model

As we described above, the general 3D graphics architecture is a pipeline consisting of multiple stages, which is represented as P = Pstage1,Pstage2,...,Pstagen. We can use a 4-tuple {Si, Pi, Ci, Ni} to represent each stage. Pi is the throughput factor which is the parallelism determined in the design time, for instance, nVidia Geforce 7800 can process 24 instructions in parallel at the same time, 24 is the throughput of the shader stage in the pipeline. Ci is the worst case execution time of the pipeline stage at the maximum processor speed. Si is a state of a graphics feature in 3D graphics which are either enabled or disabled by the APIs in the program such as shading models, lighting models, and texture modes. Ni is the iteration factor based on the number of primitives, number of vertex, number of pixels and the fragment buffer or depth buffer. Hence, the execution path and execution time of each pipeline stage can be changed depending on these features. Parallelism is

defined in the design time and means how many operations can be done at the same time. Therefore, the execution time of j_{th} frame can be stated as

$$D_j = \sum_{i=1}^{n} \frac{C_i S_i N_i}{P_i} (1 < i < n) \tag{3.1}$$

Since C and P and fixed valued defined at design time, the execution time is closely related to S and N and comes from the application's workloads. Even though the pipeline is well optimized in the design, there are still idle times due to the imbalances that occur due to differences between Si and Ni. The bottleneck for the frame is

$$B_j = max(\frac{C_i S_i N_i}{P_i})(1 < i < n) \tag{3.2}$$

which has the maximum delay time in n stages. Therefore, when it is the execution time of the bottleneck stage of the whole pipeline, the other stages can either process more inputs or have idle times. That means we can have a chance to optimize the performance via load-balancing, or reduce the supply voltage or clock .

3.2 Definition of Multiple Clock Domains (MCD)

The graphics architecture is defined having six clock domains as follows according to their functions and uses[27][32].

3.2.1 Vertex Shader Domain.

The vertex shader evolved from the standard transform and lighting stage in the fixed-function pipeline. The transform and lighting pipeline is controlled by setting render states matrices, and lighting and material parameters. Instead of setting parameters to control the pipeline, vertex shader is a program that executes on the

graphics hardware. It generates one output vertex from each vertex it receives as input which means it is neither capable of creating vertices nor writing to other vertices than the one it currently shades. With the hard-wired transform and lighting pipeline, many of the effects used in games are similar. While with vertex shader, games can create a lot of interesting effects such as procedural geometry (used in cloth simulation, bubbles etc.), vertex blending for skinning, advanced keyframe interpolation like complex facial expressions, particle system rendering or some sophisticate lighting models. Therefore, the workload for this domain varies significantly depending on which effects are created.

3.2.2 Rasterizer Domain.

Rasterizer domain incorporates primitive assembly, clipping, triangle setup and fragment generation. The main role for the rasterizer is to interpolate the properties of each fragment such as positions, normals, lightings or texture coordinates. The rasterizer interpolates a property values for each pixel from the properties of the vertices. These values are interpolated using a weighted average of the edge's vertex values, where the properties value data at edge pixels that are closer to a given vertex more closely approximate values for that vertex. Therefore, the number of the primitives and the covered areas of these primitives are the two important factors for the workload of rasterizer.

3.2.3 Pixel Shader Processor Domain

The pixel shader evolves from standard multiple stage of texture in fixed-function pipeline. It computes shading colors on each pixel. The most understandable example for pixel shader is per-pixel non-standardized lighting. In the standard multiple stages of a texture pipeline, the effects possible with the approach were very limited

to the implementation of the graphics card device driver and the specific underlying hardware. However, the visual effects gained from pixel shaders are enormous. It is widely used for implementing effects like true phong shading, anisotropic lighting, non-photorealistic-rendering, per-pixel fresnel term and self-shading bump maps. It gets the already multisampled pixels along with z, color values and texture information from rasterizer [28]. The already Gouraud shaded or flat shaded pixel might be combined in the pixel shader with the specular color and the texture values fetched from the texture map. For this task the pixel shader provides instructions that affect the texture addressing and instructions to combine the texture values with each other in different ways . While it is flexible to create desirable visual experience with the pixel shader, it is the most expensive part of computation in the pipeline since the operations are based on pixels. For instance, the shader operations need to be executed repeatedly 1280x1240 times for a full screen application with 1280x1240 resolution. Similarly, the shader program length can vary dramatically according to the complexity of the visual effects. The number of pixels to be drawn can be quite different across the frames which may affect the workload significantly.

3.2.4 Texture Domain.

Texture are generally used for mapping onto the surface of polygons. It can add details to a surface with little cost. As mentioned in the pixel shader processor domain, the previous fixed-function pixel engine is mostly multiple texture cascade and blending. The use of multiple texture blending can profoundly increase the frame rate of a Direct3D application. An application employs multiple texture blending to apply textures, shadows, specular lighting, diffuse lighting, and other special effects in a single pass. The texture sampling is a high cost operation and could stall the shader process. So it usually uses two level memory hierarchies. A

typical texture unit may have three parts - texture address unit, texture cache and texture memory. The texture address unit calculates the actual texel coordinate according to the texture address mode and the filter mode. The texture cache improves the speed to fetch the texel. When the calculated texel access cannot hit the cache, the texture backend needs to be accessed and the texture cache needs to be updated. Since the use for textures varies significantly for applications, it is helpful to have a separated domain for texture. For some applications which just exploit the powerful computation ability of graphics hardware without using any texture, the texture units can be turned off.

3.2.5 Render Buffer Domain

The render buffer is the memory to store the properties (color) of all of the visible pixels in the rendered 2D image. It is the last stage of the pipeline. The color for each pixel is a 32 bit or 16 bit value. It usually has 3 or 4 channels, R(red), G(green), B(blue) or A(alpha). The rate of writing the pixels to the render buffer is called the fill-rate, which is a very important measure of performance of graphics hardware. The operations for the render buffer include both the write operations and read operations. For read operations there are two important effects which need to read the colors of pixels from the render buffer: fog blending and alpha blending. Fog is implemented by blending the color of objects in a scene with a chosen fog color based on the depth of an object in a scene or its distance from the viewpoint. As objects grow more distant, their original color blends more and more with the chosen fog color, creating the illusion that the object is being increasingly obscured by tiny particles floating in the scene. A fog factor is computed and applied to the pixel using a blending operation to combine the fog amount (color) and the already shaded pixel color, depending on how far away an object is. The distance

to an object is determined by its z- or w-value or by using a separate attenuation value that measures the distance between the camera and the object in a vertex shader. If fog is computed per-vertex, it is interpolated across each triangle using Gouraud shading. Alpha blending is used to display an image that has transparent or semi-transparent pixels. In addition to a red, green, and blue color channel, each pixel in an alpha bitmap has a transparency component known as its alpha channel. The alpha channel typically contains as many bits as a color channel. For example, an 8-bit alpha channel can represent 256 levels of transparency, from 0 (the entire pixel is transparent) to 255 (the entire pixel is opaque). Alpha blending combines the source color from the transparent object and the destination color (the color already at the pixel location). The read and write operations to and from the render buffer usually cost more cycles than normal computing instructions. Thus, an independent voltage and frequency setting for the render buffer is important.

3.2.6 Depth Domain.

The depth domain is divided into three stages: hierarchy z, z testing and depth buffer. The depth buffer is a high traffic part of the graphics hardware since we need read back depth from the buffer and do the depth test and write back the passed depth value to the depth buffer. Read and write operations are always expensive. Therefore, modern graphics hardware usually implements hierarchical z with two levels or three levels of memory hierarchy. The smaller the cache, the higher level in the hierarchy. Z cache usually stores the range of z value(zmin, zmax) for a block of pixels. Only the pixels with z value inside the range can be processed by the following parts in the pipeline. The depth buffer stores the depth value for each pixel. The z (depth) test compares the current calculated depth value with the value stored in the depth buffer. According to the comparison function such

as less, or greater, the tests fail or pass. For the pixels with pass results, they are going to be written to the render buffer and be seen from the screen. At the same time, the new depth values for those pixels are updated to the depth buffer. On the other hand, if depth tests fail, the pixels are not going to be written to the render buffer and the depth buffer is not going to be updated for those pixels. Since the computations on pixels are expensive, we would like to use the z test unit before the pixel shader units to reject those pixels which are not going to be output to the render buffer and depth buffer. Z unit is a complex logic unit inside graphics hardware and there are many operations on it and also many physical logic units inside the hardware. When applications are not using the depth buffer at all, the depth domain can be shut off and significant energy could be saved.

3.3 Implementation in Simulator

We implement our multiple clock domain architecture in Qsilver. We depict how these domains work inside the simulator in detail as follows.

- **Render Buffer Domain**: We simulate the render buffer as a buffer pool. When the pixels are written to the render buffer, it will increase operation counters in the framebuffer operation units, the post-pixel operations like alpha blend, fog blend, alpha test, z test and stencil test are done on the unit, and drains the queue between the unit and the fragment processor. In each cycle, the maximum number of pixels to be written to the render buffer is limited to the bandwidth for the simulated hardware. In our case, we simulate the Nvidia Geforce 7800GTX processor, whose fill rate is 61.9GB/s and the memory frequency is 500MHz. The maximum numbers for pixel written to the render buffer is 32 for a pixel with 32 bits. When the application is bound

to the render buffer, the shaders are extremely simple and all pixels which need to be written to the render buffer are visible. It can only take 32 pixels in our simulator for an adapter like Geforce 7800GTX to write to the render buffer while there are 96 pixels finishing the fragment processing and ready for write. The write queue is full and no more pixels can come to the queue . Thus the pipeline is stalled waiting for the fragment buffer. In this case, when we increase the voltage or the clock of frequency, it may help reduce the writing latency to the render buffer and improve the performance for the entire application. On the other side, when there are many computations for each pixel, it may take numerous cycles to finish the fragment process and to output to the queue. Hence, the queue between the fragment operation and the render buffer is empty. The written render buffer unit may need to wait for the data coming to process. In this case, the voltage and the frequency for the render buffer fill-in can be lowered to save the power and further save energy. As a result, the energy can be saved by scaling of voltage and frequency.

- **Depth Domain**: The depth domain is similar to the render buffer domain in that it holds its own voltage and frequency scaling factors. When the pixels finish the execution cycles of the computation operations, they are written to the depth buffer at the same time as the render buffer. We don't simulate hierarchy z here and the discussion of the design policy for the multiple level z cache is outside the scope of this thesis. The cycles for doing operations with the depth buffer can be zero which means depth write disable for the application and the depth domain can be shut off.

- **Pixel Shader Domain**: The pixel shader domain is implemented as a function unit which drains a queue of fragments interpolants such as positions

and colors filled by the rasterizer. Fragments are potential pixels which are interpolated from the vertices of the polygons. Fragments that pass the depth test, the stencil test or the alpha test following the pixel shader unit are pixels written to the fragment buffer and displayed to the screen. The number of fragment interpolants varies on the applications, which may include colors for diffuse light, colors for specular light, texture coordinates, positions, normals or fog colors. The function unit will run a number of cycles on a block of fragments. When the shader processor is ready to produce a fragment (a 2x2 arrays of fragments in our model), the processed fragments are added to the queue. The rate of the processor depends on the computation demanded by the shader instructions, the texture access rate and on the intrinsic architectural parameters of the processor such as the number of the shader processors or the number of texture units. For the NVidia GeForce 7800GTX, it has 24 shader units and 16 texture units. In a given cycle, the shader processor may stall for several reasons: the subsequent queue may be full, which generally implies the application is framebuffer bound, the texture fetches cannot be performed, which means the application may be texture-bound or it has a poor texture cache algorithm, or because of the incoming queue is empty(rasterization bound). As the shader processor produces fragment, the counters for the shader instructions are collected and the power for executing the amount of instructions are summed up. The shader processor will drain a queue, the queue of fragment interpolants (position, lights, normal etc.). A full or empty queue implies that the front end (rasterization, triangle setting up) or shader processor is stalled.

- **Texture Domain**: The texture domain is attached to the shader processor. It processes texture requests for a whole fragment quad. The number of texture

accesses is passed from the trace file. This number depends on several factors: how many textures are bound, the form of texture filtering enabled, and the per-pixel texture Level-of-Detail(LOD) of the fragments. In our model we don't implement the texture cache and the cycles for texture lookup are the cycles to fetch a texel from the texture memory. Current graphics hardware usually implement texture cache and the cost for lookup depends on the hit-rate of the texture cache.

- **Rasterizer Domain**: The rasterizer is implemented as a function unit which drains a queue of vertices filled by the vertex shader; In the function unit, the input is the queue of vertices and the output is the queue of fragments. The computational operations executed by the rasterizer are derived from the number of fragments generated per primitive and the number of active interpolants (from the trace files). The computation results from the rasterizer are added to the queue which is the input for the next stage. The rate of the rasterizer depends on the number of polygons or vertices (usually graphics hardware process polygons as triangles formed with three vertices), the number of fragments, the area inside the polygons and the number of interpolants. Therefore, for the application which generates an almost constant number of primitives while the generated area varies substantially across frames, the rasterization part has a variable load.

- **Vertex Shader Domain**: The vertex shader domain is implemented as function units which execute computations on each vertex input from the pre-vertex cache. The cache is filled according to the data coming from the trace file. As the vertex data fetched from the cache is ready, the counters for the shader instructions are collected and the power for executing the amount of

instructions summed up. The units then fill in the queue which is going to be drained by rasterizer domain.

3.4 Development of Energy-friendly Application Program Interface

We argue that applications know best about their resource and energy needs. Applications can implement better power saving policies than the generic utilization based approaches previously proposed on consumer graphics hardware or the driver from graphics hardware vendors[22]. For instance, for general purpose GPU applications, most of the operations are done in shader execution units and texture units. Then the applications just need to turn off the other four clock domains to save energy. Currently there are no defined energy-friendly APIs in the graphics literature or specification of popular graphics libraries such as OpenGL or DirectX. Inside graphics hardware, there are two clock domains, one is for the execution engine and the other is for GPU memory. Tools such as rivaTuner [30] allow users to adjust GPU clocks system wide. However, this is a static setting done once before the application is run. We propose an API to set voltage and frequency scaling factors dynamically at run time based on application behavior.

To measure the benefits of our proposed API, we adopt an openGLES[15] library, which can generate code that mimics OpenGL. As previously mentioned, Qsilver uses Chromium [13] to obtain trace and statistics data as input to the graphics simulator. To get the trace for an OpenGL application, the openGLES library is modified to intercept the commands and data passed by the applications. It saves the commands and data for future change and sends them back to real graphics hardware to do the original application. On Windows, this is accomplished by

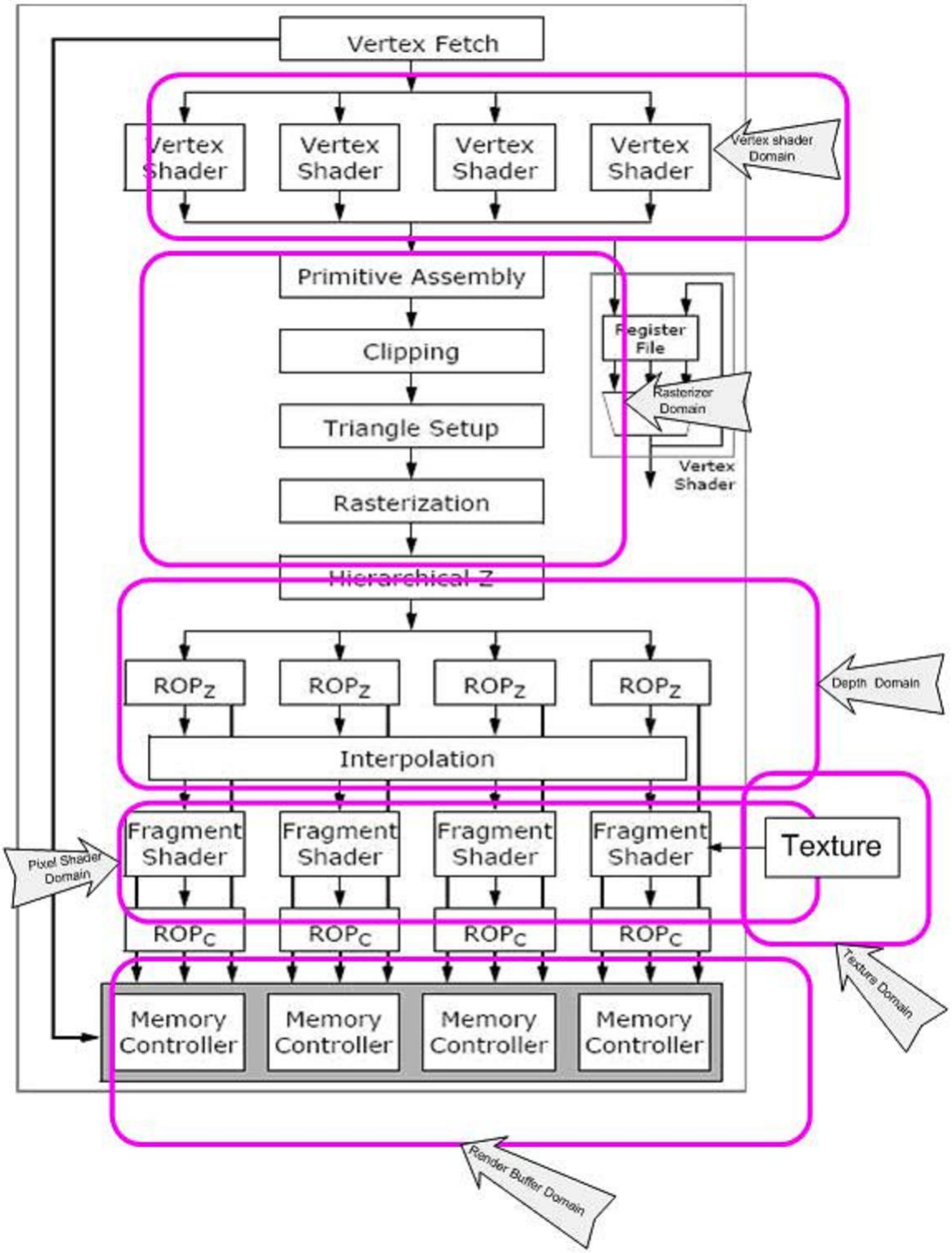

Figure 3.1: Multiple Clock Domains

creating a temporary directory, copying the executable there, copying libGlESV2.dll to that directory , spawning the executable as a child, and deleting the directory when the child exits. Then we incorporate our APIs into this wrapper and generate the clock and voltage information to the trace file passed to the Qsilver system for analysis. For each glSwapBuffer, we record the API to the tracefile which will be identified in the simulator. In this way we create the power-friendly APIs which do not exist currently.

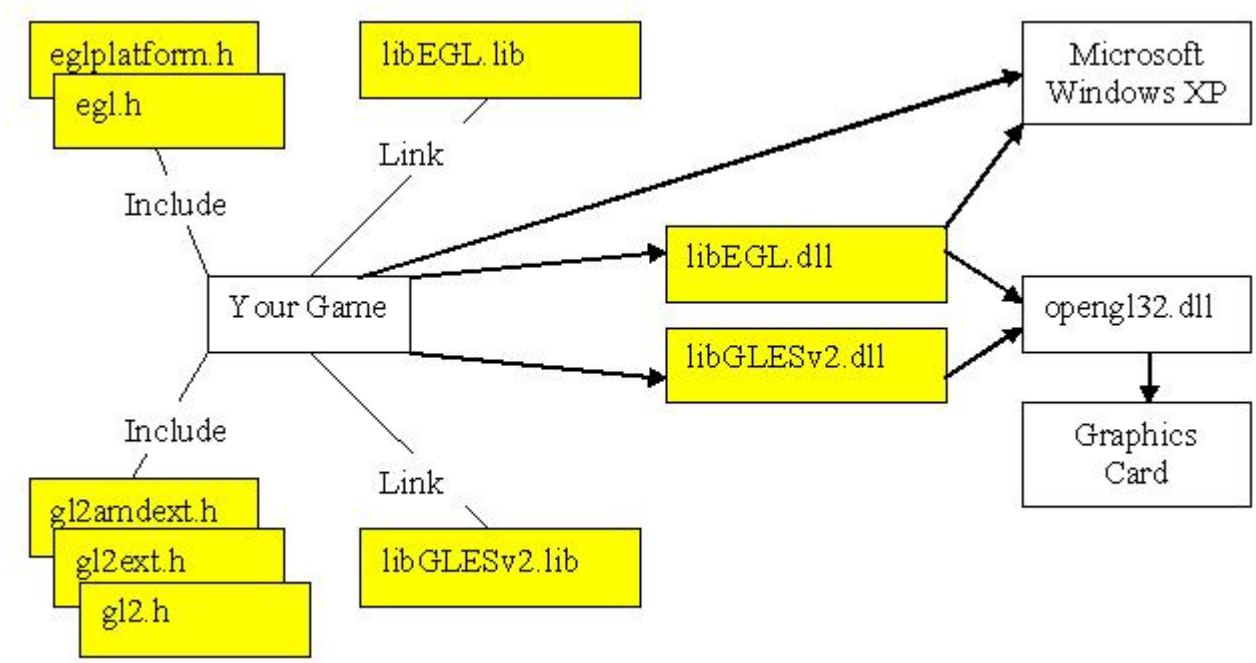

Figure 3.2: The Architecture of the Application-Directed DVFS Framework

We define the interface for setting the scaling factor which is the percentage of full speed as follows. glSetMCDScalingFactor sets the scaling factor for each domain. glMCD enables or disables MCD.

```
glSetMCDScalingFactor(MCDFactorSetting);
glMCD(Enable/Disable);
```

We define MCDFactorSetting as a structure as in Figure 3.3. We use this API in our tracefile to notify the scaling factor setting.

```
struct MCDFactorSetting
{
MCD_fb_factor;
MCD_z_factor;
MCD_vs_factor;
MCD_rastrizer_factor;
MCD_texture_factor;
MCD_ps_factor;
}

/***********************************************
/
/***************Example Useage***************/


...

glMCD(GL_ENABLE);
MCDFactorSetting mcd;
mcd.MCD_fb_factor = 1.0;
mcd.MCD_z_factor = 1.0;
mcd.MCD_vs_factor = 0.25;
mcd.MCD_rasterizer_factor = 1.0;
mcd.MCD_texture_factor = 1.0;
mcd.MCD_ps_factor = 1.0;
glSetMCDScalingFactor(mcd);
```

Figure 3.3: Example of MCDSetting Structure

Compared to the application directed voltage and clock configuration, another way for dynamic voltage and frequency scaling is to detect the data throughput and adjust the configuration when new phases are detected. When the data throughput is lower or higher than a threshold, the corresponding configuration is applied. Otherwise the original configuration will be restored.

To compare the results of our application-driven policy with the throughput detecting policy, we also implement a dynamic scheduling algorithm inside the simulator. The dynamic scheduling algorithm detects the throughput between each domain by using the queues decoupling stages. When the queue is full, this means the GPU is next stage bound, the previous stage in the pipeline is effectively stalled and it should reduce its voltage and frequency. Likewise, when the queue is empty, the GPU is bound in the previous stage and the next stage is actually stalled and should reduce its voltage and frequency. While with the change of voltage and frequency, the trend for the bounding part could be changed and moved to another domain. We set four threshold for each queue to identify full or empty, low low, low high, high low and high high (see Figure 3.4). The low low means the queue is near empty. The high low means the queue is slightly lower than half. The high low means the queue is more than half. And the high high means it is near full. When the queue is detected as mcdLl, it means the previous stage is bound and the next stage is stalled, and should reduce its voltage and frequency. When the queue is detected as mcdHh, it means the next stage is bound and the previous stage is stalled and we should reduce its voltage and frequency. Whenever the mcdLh or the mcdHl status is detected, this means the throughput is fine and the scaling factors should get back to the full speed. We implement the detection policy and compare it with the application directed policy in Section 6.2.

```
qu = queue_utilization(tile);

if (qu <= (((double) get_config(mcd_low_water_l)) / 1000.))
  mqs = mcdLl;
else if (qu <= (((double) get_config(mcd_low_water_h)) / 1000.))
  mqs = mcdLh;
else if (qu <= (((double) get_config(mcd_high_water_l)) / 1000.))
  mqs = mcd_stable;
else if (qu <= (((double) get_config(mcd_high_water_h)) / 1000.))
  mqs = mcdHl;
else
  mqs = mcdHh;
...
```

Figure 3.4: Polling Policy for MCD Setting

Chapter 4

Workload Prediction Algorithms for DVFS

4.1 Interval-based Algorithms

Many workload prediction algorithms for DVFS (Dynamic Voltage and Frequency Scaling) are proposed in recent years. However most of them are targeting for general-purpose computer systems or real-time and embedded system. They usually make use of the imbalance of the workload between the components of the computer systems like processors and memories.

While in our work, we use dynamic voltage and frequency scaling for graphics hardware with two further innovations. First, we use multiple clock domains where each stage in the graphics pipeline uses a different voltage and frequency. Second we use history-based workload predictors for 3D graphics to set voltage and frequency.

An interval-based algorithm sets scaling points at fixed intervals (for instance, 1 frame) and calculates a scaling factor for each interval. In general it predicts how busy each stage in the GPU will be in the interval and then sets the voltage and

frequency for each stage accordingly. At each interval, the processor utilization for the interval is predicted, using the utilization of the processor over one or more preceding intervals.

Two prediction schemes have been used in interval-based scheduling: the moving-average and the weight average schemes. In the moving-average scheme, the next workload is predicted based on the average value of workloads during a predefined number of previous intervals, called window size. In the weight-average scheme, a weighting factor, α,is considered in calculating the future workload such that sever fluctuation of the workload is filtered out, resulting in a smaller average prediction error. Their operations are represented in the following equations.

$$WindowSize = n$$

Moving-average:

Workload(t+1)=$\sum_{i=1}^{n-1} \frac{Workload(t-i)}{n}, t >= n-1$

$=\sum_{i=0}^{t} \frac{Workload(t-i)}{t+1}, (4.1)$

$$Weight - average:$$

Workload)(0) = Workload(0)

Workload(t+1)=$\alpha Workload(t) + (1-\alpha)Workload(t)$

$=\sum_{i=0}^{t} Workload(t-i)(1-\alpha)^i, (4.2)$

In scheduling the voltage at which a system operates and the frequency at which it runs, a scheduler faces two tasks: to predict the future system load (given past behavior) and to scale the voltage and clock frequency accordingly. These two tasks are referred to as prediction and speed-setting. We consider one algorithm better

than another if it finishes the same task as another policy but with less energy consumed.

We consider two predictions algorithms PAST and AVGN originally proposed by Weiser et al[42]. Under PAST, the current interval is predicted to be as busy as the immediately preceding interval, while under AVGN, an exponential moving average with decay N of the previous intervals is used. That is, at each interval, we compute a "weighted utilization" at time t, Wt as a function of the utilization of the previous interval U_{t-1} and the previous weighted utilization W_{t-1}. The AVGN policy sets

$$W_t = \frac{NW_{t-1} + U_{t-1}}{n+1}. \tag{4.3}$$

The PAST policy is simply the AVG0 policy, and assumes the current interval will have the same resource demands as the previous interval. The decision of whether to scale the clock and/or voltage is determined by a pair of boundary values used to provide hysteresis to the scheduling policy. If the utilization drops below the lower value, the clock is scaled down; similarly, if the utilization rises above the higher value, the clock is scaled up. [36] set these values at 50% and 70%. We used those values as a starting point, but deciding how much to scale the processor clock is separate from the decision of when to scale the clock up (or down). Since PAST is actually the special case of AVG0, we have considered interval-based algorithms as follows by combining algorithm AVGN with MCD (Multiple Clock Domain).

Static Uniform The voltage and frequency of each domain are all set to the same value based on perfect knowledge of future workloads. The voltage and frequency are set to the maximum or minimum of the three stages.

Static Non-uniform The voltage and frequency of each domain don't change through the entire application, while the setting value for each domain could be

different. For instance, for a general purpose GPU application, the fragment shader unit is always the most work intense part. The voltage and frequency for this domain can be set as the highest value while the other domains can be set to a lower level. This setting needn't change since it goes through the entire application.

Dynamic uniform AVGN The voltage and frequency of each domain change through the entire application, while the setting for each domain is the same. The workload of each domain in the current frame is predicted to be the average of workloads in the previous N frames.

Dynamic Non-uniform AVGN The voltage and frequency of each domain change through the entire application, and the setting for each domain are different based on the varied workloads on each domain. The workload of each domain in the current frame is predicted to be the average of workloads in the previous N frames.

4.2 Signature-based Algorithms

Signature-based algorithms use signature to detect working set changes and trigger a tuning[24]. Signature is a concise structure which records the important information to describe the workload of the current frame. We use relevant information which is available from the 3D graphics pipeline through the APIs (openGL ES extensions) to construct a concise signature based on a sample interval.

We use a global table to record the signatures and the corresponding working set. When the signature is first generated, it is inserted into the signature table, along with the conservative workload prediction and the minimum voltage and frequency setting. After the frame is rendered, the real workload can be determined from performance counters for each domain and updated to the signature table.

Although the signature table is kept as concise as possible, the lookup and com-

pare the signatures still have performance cost. To avoid unnecessary lookup and comparison operations , we define a noise value and the stable/unstable status. We don't want to lookup and compare the signature for each frame. Only when the deviation of the current signature is beyond the noise, is the status changed from stable to unstable. When the status is identified as unstable, signature table lookup and the comparisons are done to find the best match. The smallest distance metric with the signatures from the signature table is used to define the best match. Specifically, for a generated signature C which comprise arguments $c_1, c_2, ..., c_n$ and a signature S which has arguments $s_1, s_2, ..., s_n$, the distance metric D(C,S) is

$$D(C, S) = \sum_{i=0}^{t} \frac{(C_i - S_i)^2}{C_i}; \tag{4.4}$$

The distance metric is linear and normalized to the generated signature. We introduce two statuses here: stable and unstable. For the status of stable, which means there are minor differences between the frames. This is important for the 3D graphics applications since it is really common that there is no big changes for the consecutive scenarios and frames have very similar workloads.

When the status is unstable, we search the signature table to find the best match, namely the one with the minimum distance metric. With the match signature, the corresponding power setting can be obtained. If we cannot find the match signature in the signature table, we will insert a new one in the signature table and fill in our workload prediction.

The initial state is unstable and the initial clock and voltage setting are chosen to be the minimum. If the frame rate is less than the standard one, we will increase the setting by a fixed step. Meanwhile we will insert the signature in the signature table. This exploration continues until the maximum clock or voltage is reached or until

the frame rate is good enough without noticeable pause of the frames (the frame rate per second no lower than 25fps). At this point the system state is switched to the stable status. The system remains in the stable state while the signature does not significantly differ from that in the previous interval. When there is a change, the system is switched to the unstable state and the system returns to the minimum configuration and the algorithm starts exploring again. The pseudo code for the mechanism is listed in Figure 4.1.

Instead of using a single fixed number as the noise to detect phase changes, we change this dynamically. If an exploration phase results in picking the same cache size as before, the noise threshold is increased to discourage such needless explorations. Likewise, every interval spent in the stable state causes a slight decrement in the noise threshold in case it had been set to too high a value.

To estimate the workload, we record the time (cycle counts) between two frame intervals. When the applications go from one frame to the next one, it calls function like swapBuffer in OpenGL to swap the front buffer and the back buffer. We record the time interval between two swaps. This is the execution time for the frame and this execution represents the workload. This is the actual workload and used to refine the workload of the signature in the signature table. For our simulations, we used the cycle counts spent by each frame to represent the workload.

At the end of each frame, we generate a signature for this frame. The signature is based on the triangle numbers, pixels, vertex counts, texture size, render target size, shader instructions numbers. The signature is constructed using actual values of these six parameters, concatenated into a string. These six parameters not only roughly represent the workload of the current frame but also are related to the inherited six clock domain we define. Based on this information, the selection mechanism picks one of two states-stable or unstable. The former suggests that

behavior in this interval is not very different from the last and we do not need to change the voltage and clock configuration, while the latter suggests that there has recently been a phase change in the program and we need to explore and pick an appropriate setting for the voltage and clock configuration.

We introduce our signature based work prediction algorithm as following.

Dynamic NON uniform Signature Voltage and frequency of each domain change through the entire application, and the setting for each domain are different based on the varied workloads on each domain. The workload of each domain in the current frame is predicted to be as same as the previous one which has the same signature as the current one. Figure 4.1 illustrates the proposed signature-based workload prediction algorithm.

In this chapter, two kinds of workload prediction algorithms are described: interval-based and signature-based. Interval-based algorithm predicts the workload as previous interval while signature-based algorithm predicts it as the one with matched signature. Both of the algorithms are considered to apply to the DVFS system with multiple clock domains. The experimental results for these algorithms are shown in Section 6.3.

```
generateSignature;

If ( state == STABLE )
   If ( counter < noise )
         Decrease noise
   Else
         State = UNSTABLE

If ( state == UNSTABLE )
   if (signatureMatch( signatureTable))
         pickup setting from the signature table with the best match signature
   else
         recordSignature;
         Frequency/voltage = SMALLEST;

         if (FrameratePerSecond<Threshold)&&(Frequency/Voltage!=MAX
            TUNING(Frequency, voltage);
         state=STABLE;

if setting==previous_setting)
   increase noise;
```

Figure 4.1: Signature-based Algorithm

Chapter 5

Experimental Environment

The chapter describes software setting, the trace file, the simulator and the measurement metrics for the tests. The test scenes and the methods used to measure power and time are presented here as well.

5.1 Software Model

5.1.1 The Trace

We use the states and iterate numbers (the numbers of pixels, textels, instructions etc.) as a trace to pass through the Qsilver simulator which simulates the stages in a general 3D graphics Pipeline. Driving the simulator is a trace of graphics parameters and the corresponding statistics data such as how many geometries or texture. We adopt a 3D software library that implements OpenGLES, a standard API for implementing 3D applications on embedded platforms. We modify the library to generate the trace of how many triangles, vertices, pixels, instructions and resource usages for each frame. The trace then is passed to the modified Qsilver simulator to analyze the performance and energy consumption for each domain.

5.1.2 Simulator

To evaluate the benefits of our DVFS algorithms and MCD, we conducted experiments on an extensively modified version of the Qsilver simulator. The original simulator was extended by more detailed functional units and support for MCD and DVFS.

Qsilver [39] is a queue driven, cycle accurate graphics hardware simulator. Data flows through decoupled stages which act like functional units inside graphics hardware. Each stage is a functional unit inside the hardware. Between each stage, there is a queue (FIFO) to maintain the data coming out from the previous stage and being used as the input to the next stage. The cycles to execute a shader instruction or memory access instruction are compared to the ARM processor on similar frequency and gate counts. The input to the simulator is a trace which includes graphics parameters and their corresponding statistics data such as how many geometries or texture. For instance, for a triangle rendered, the trace may include the numbers of vertices, the pixels inside the triangle to be rendered, the numbers of light, or the number of textures . The number of operations and the computation load for each functional unit will be counted on the cycle-timer model after the trace passes through the simulator.

The simulator executes the trace one cycle by one cycle. It repeatedly advances a global time counter by one cycle and advances the simulation stage by stage through the pipeline. The traces flow through each stage in the pipeline, do the operations, wait for the queue drain (when the queue is full) or wait for data coming (the queue is empty) on each cycle.

We modified the simulator and implemented the new features. We implement six clock domains and every one has separate frequency and voltage settings. We have

two ways to control the settings. One is that we define a low and high threshold for the data throughput for each domain. When the throughput is beyond the threshold, the clock and voltage setting will go the next level. Another way for control setting is from the API which can read from the trace to visibly set the clock and voltage levels. In addition, we add the computation workload estimation to programmable pipelines. Number of instructions and the cost of both cycles and energy consumptions are considered and counted on the DVFS system.

5.2 Comparison Metrics

We use the total execution time T, the average power consumption P the total energy usage E, and the energy-delay product ExT [3] for comparison. All results are relative to the same program running at the peak frequency. As mentioned in Chapter 1, power and performance trade-offs motivates the technique of dynamic voltage and frequency scaling. While an application can be executed with low power, its execution time may be unacceptably long. It is insufficient to simply look at average power, because E=PT. Hence, if reducing average power comes at a cost of increased execution time, more energy may actually be consumed. To include the latency constraint, energy and energy-day product are used by many as the metrics for the comparison of different power-aware techniques. The energy is equal to the product of the average power consumption and the total execution time. Energy-delay product is equal to the product of the energy usage and the total execution time, i.e., $ET = PT^2$. Energy translates directly to the battery life, and energy-delay product ensures a greater focus on performance.

5.3 Power Model

We estimate the power by counting on the power for each operation like shader instruction, texture cache access, depth cache access, and texture fetch and memory access to depth buffer and render buffer, etc. By identifying all of the operations for each domain and applying the scaling for each domain, we can construct an estimate for the energy consumed by the application.

We extract the cost of similar operations from an industrial architecture-level power model to estimate the power for each operation[40]. This model is based on circuit-extracted data for an 180nm high-performance superscalar microprocessor. We then scaled these estimates to match the structure sizes and bit-widths used in our simulator, and further scaled them to the appropriate technology node according to cv^2f.(power is proportional to cv^2f,where c is capacitance, v is voltage, and f is frequency.) In our case, the current simulator resembles an Nvidia Geforce 7, so we have chosen to model a 90nm implementation running at 1.8v and 500MHz. Although there will certainly be differences in circuit-design style between a high performance CPU and GPU, the relative power cost among different operations in the high-performance CPU is likely to be a reasonable indicator of relative cost in the GPU. The more important thing is that we compare the internal energy consumption using the GPU simulator with different DVFS algorithms.

Unlike DVFS, MCD employs voltage and frequency scaling at a sub-chip granularity. The six domains are rasterization, depth process, post shader, fragment buffer, texture and shader execution. Each domain can be set as idle, static or dynamic. If a domain is idle the unit inside the domain will be shut off to save energy. Static means that domains are clocked at a fixed voltage and frequency. If a domain is dynamic, the right voltage and frequency will be set from the commands passed

by applications.

We implement our multiple clock domain architecture on Qsilver. In our system, the power consumption for each domain is equal to the power consumed for operations running on the unit plus the power for idle status.

$$E_t = E_r + E_i. \tag{5.1}$$

Et is the total energy consumed. Er is the energy consumed by the operations running on the units and Ei is the energy consumed when the domain is idle. Some units can be idle when they finish processing data passed to them or while they are waiting for the input data coming from previous units. In these cases, these units do not run at peak and their frequency and voltage can be lowered to save energy. Conversely, some units may be busy and always run at peak. In this case the voltage and frequency on these units can be increased to reduce bottlenecks in the units and speed up the entire applications and also save overall energy consumption.

The examples for power model for each stage are given in Figure 6.1. Basically, the power for each domain includes several parts: the cost for outputting to the next queue, the cost for reading from the previous queue, the cost for operations in a specific stage, as well with the scaling factor for the domain.

5.4 Test scenes

To discuss the workload variation and prediction algorithms, we adopt some widely used samples from RenderMonkey[1]. RenderMonkey is a free downloadable environment for Direct3D, openGL and openGLES. It is targeted to both programmers and artists. RenderMonkey also includes a large number of sample effects which are freely available to be used and customized for 3D graphic applications. We analyze

```
add_to_stage_power(rasterize_vertices,
(((get_period_counter(tl_queue_writes) *
get_config(tl_queue_write_cost)) +
(get_period_counter(tl_queue_reads) *
(get_config(tl_queue_read_cost) +
/* Vertex setup cost */ 0)) +
((get_period_counter(fragments_created) +
get_period_counter(texcoord_interps)) *
(get_base_power(scalar_fmul) +
get_base_power(scalar_fadd)))) +
idle1 *
get_dynamic_max(rasterize_vertices)) *
mcd_power_scaling_factor(stage_rasterize_vertices) *
dvs_scaling_factor(get_current(dvs)));

add_to_stage_power(process_fragments,
(((get_period_counter(tile_queue_writes) *
get_config(tile_queue_write_cost)) +(get_period_counter(tile_queue_reads)
*
get_config(tile_queue_read_cost)) +
(get_period_counter(texture_interps) *
((3 * get_base_power(scalar_fmul)) +
(2 * get_base_power(scalar_fadd)))))) +
(((double)idle2) /
((double) (get_config(tile_pipelines) *
get_config(fragments_per_tile)))) *
get_dynamic_max(process_fragments)) *
mcd_power_scaling_factor(stage_process_fragments) *
dvs_scaling_factor(get_current(dvs)));

add_to_stage_power(write_framebuffer,
(((get_period_counter(fb_queue_reads) *
(get_config(fb_queue_read_cost))) +
(get_period_counter(fb_queue_writes) *
get_config(fb_queue_write_cost)) +
(get_period_counter(fragments_z_tested) *
(get_base_power(scalar_fcmp))) +
(get_period_counter(fragments_z_passed) *
get_config(fb_queue_write_cost))) +
idle3 *
get_dynamic_max(write_framebuffer)) *
mcd_power_scaling_factor(stage_write_framebuffer) *
```

Figure 5.1: Examples for Power Model

```
add_to_stage_power(texture_cache,
(((get_period_counter(texture_reads) *
get_config(texture_cache_read_cost))) +
idle4 *
get_dynamic_max(texture_cache)) *
mcd_power_scaling_factor(stage_texture_cache) *
dvs_scaling_factor(get_current(dvs)));
}

double mcd_power_scaling_factor(stage_enum s)
{
fp.f_fact = fp.f_fact * get_config(mcd_ps_rate,float);
if (get_config(mcd,bool)) {
if (s == stage_process_fragments_default)
return (fp.v_fact * fp.v_fact * (fp.f_fact /
((double) get_config(clock_frequency,double))));
}
```

Figure 5.2: Examples for Power Model

the workload characteristics and evaluated the efficiency of the proposed approach using openGLES sample workspaces from RenderMonkey. The detailed discussions of each scene are given by Section 6.1.

We examine the efficiency for energy of multiple clock domains for a static scene using a representative general-purpose non-graphics application, and discuss the implications on dynamic scaling techniques for multiple clock domains. For this study, we consider a GPU implementation of ray-tracing. The application involves sixteen triangles and 64x64x4 pixels. There is only one light and no texture. The shader instructions are all pixel shader instructions and 426 in total. We use this application as a test example since it is a very typical general purpose GPU application. At the same time, it is a simple model from graphics hardware view and helps identify the usage for each domain. The tested scene is given as Figure 6.8.

Chapter 6

Experiment Results and Discussion

The purpose of our study is to introduce a prototype of the application-directed DVFS model using multiple clock domains on a hypothetical GPU and show how some workload prediction algorithms can be applied to a GPU. We examined a number of algorithms, most of which are variants of the AVGN policy. Our intent was to argue that applications know best about the resource usage and it is better to control the power-related environments directly by the applications. We split the discussion of our results into three parts. The first section describes how workload varies for multiple clock domains. The second section illuminates how DVFS for multiple clock domain works to save the energy and the third section discusses the performance of the different algorithms for workload prediction. Finally, we examine the benefit of applying DVFS to MCD and summarize the results.

6.1 Unbalancing Workload in MCD

We provide a detailed quantitative workload characterization of the collected 3D applications for the workload variance between each domain in the same frame and also the variance between different scenes for the same domain. For each domain we determine if it is used by the application and collect several statistics. Such statistics can be used to guide the development of mobile device architectures.

We use the framework discussed in Chapter 3 with Rendermonkey to compare the workload across the multiple domains. We collect several statistics for a set of scenes to depict the workloads which varies significantly across the domains.

Ambient Occlusion (Figure 6.1) shows the use of an ambient occlusion map which was computed offline using ATI's Normal Mapper tool [2]. This scalar map encodes self occlusion and is used to modulate the ambient cube map which was made with HDRShop[2] using the Grace Cathedral environment map as an input.

Anisotropic (Figure 6.2) This scene presents anisotropic shading technique. Satin effect shows how rotating tangent vectors in pixel shader to achieve unconventional reflection look. Brushed Metal effect uses gradient texture as a lookup table and compute strand lighting to achieve brushed metallic look.

Bounce (Figure 6.3) This indicates animation technique done by shaders. Based on time value, ball's position is modified to make ball look squashed.

Clipplane (Figure 6.4) This scene depicts the effect of four user-defined clip planes. The clip planes are defined by the ClipPlaneNormal arrays. The XYZ components of the arrays correspond to XYZ components of the plane's normal; the W components corresponds to the distance that the plane is away from the origin.

Depth Complexity (Figure 6.5)

This effect illustrates how additive blending can be used to display the "complex-

ity", or overdraw, of a model. One pass is used to calculate the "complexity" of the model, and the second pass is used to translate the "complexity" into a more visible spectrum. Toggling the cull mode provides additional insight into the benefits of back-face culling.

Depth of Field (Figure 6.6) This scene demonstrates a method for creating depth-of-field effects. The user is able to change the focal plane in the effect, changing the depth that will appear in focus to the viewer. The effect is generated by interpolating between an out-of-focus texture, and the in-focus texture, all based on the pixel depth.

Disco Lighting (Figure 6.7) uses a cube-map to describe the disco light. This effect renders a rotating colored light onto surrounding surfaces. The lighting for the surrounding surfaces is calculated by rotating the light-object vector by the appropriate disco rotation matrix, then looking up the resulting value from the disco light cube-map.

We present some general statistics of the workloads for the domains.

Vertex Shader ALU cycles : The total ALU cycles used for the vertex shader domain. It is equal to the shader length multiplied by the number of vertices.

Pixel Shader ALU cycles The total ALU cycles used for pixel shader domain. It is equal to the shader length multiplied by the number of pixels.

The numbers of pixels and vertices . As discussed in chapter 3, the rasterization cost depends on the number of vertices (polygons) and the fragments (areas) linearly. Therefore we adopt the number of vertices and fragments to represent the cost for rasterization.

Bytes of render buffer memory used the render buffer memory access numbers

Bytes of depth buffer memory used the depth buffer memory access numbers

Bytes of texture buffer memory used the texture buffer memory access num-

bers.

Figure 6.1: Ambient Occlusion

Detailed Workload Statistics

One important aspect for 3D graphics benchmarking is to determine possible bottlenecks in a 3D graphics environment since the 3D graphics environment has a pipeline structure and different parts of the pipeline can be implemented on separate computing resources such as general purpose processors or graphics accelerators. Balancing the load on the resources is an important decision. Bottlenecks in the vertex shader part of the pipeline can be generated by applications that have a large number of primitives with small size, which causes substantial geometry computation load. Bottlenecks in the pixel shader part are usually generated by fill intensive applications that are using a small number of primitives where each primitive covers a substantial part of the scene. An easy way to determine if an application is pixel shader intensive is to decrease the resolution and determine the speed up.

The test scenes were chosen to stress various parts of the pipeline. For instance

Figure 6.2: Anisotropic

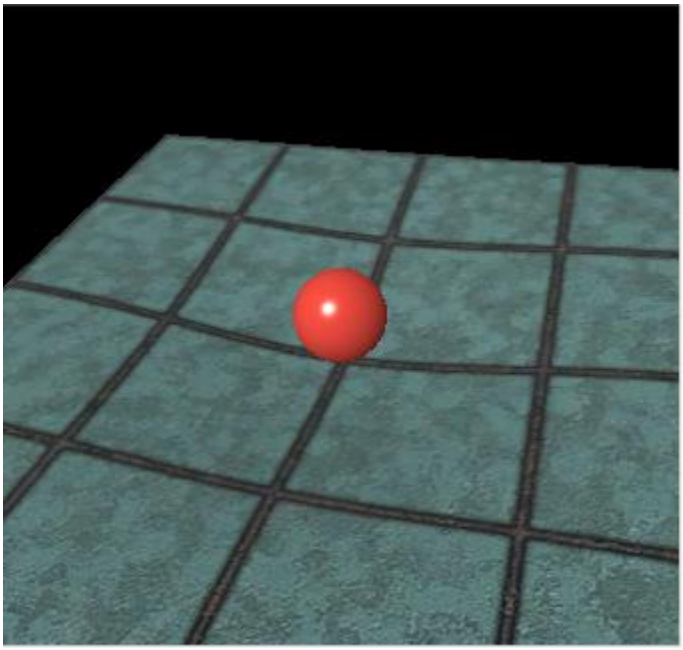

Figure 6.3: Bounce

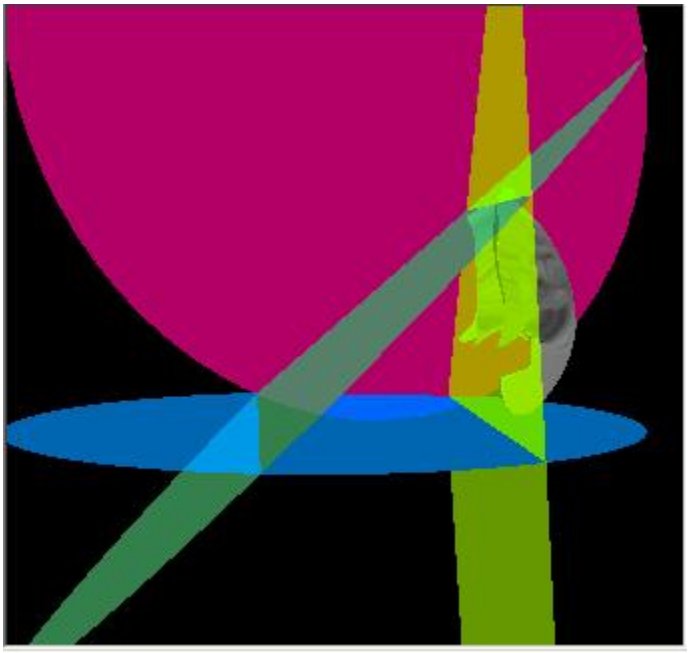

Figure 6.4: Clip Plane

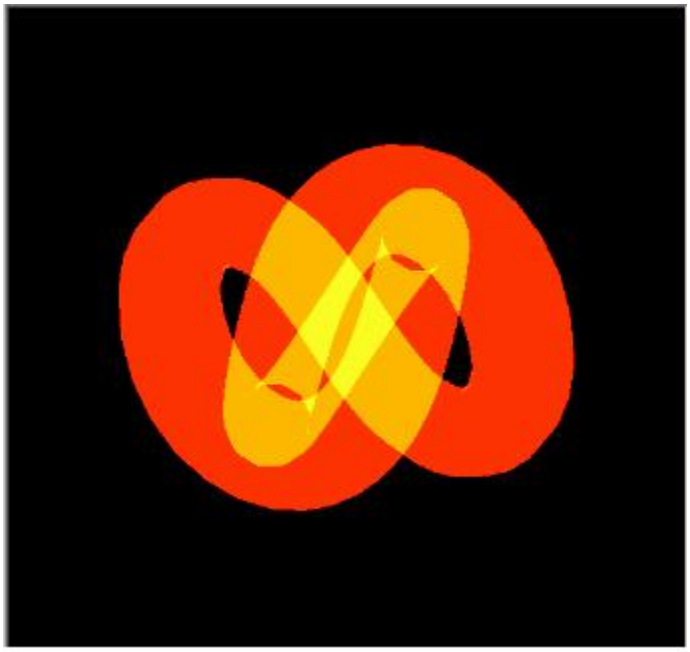

Figure 6.5: Depth Complexity

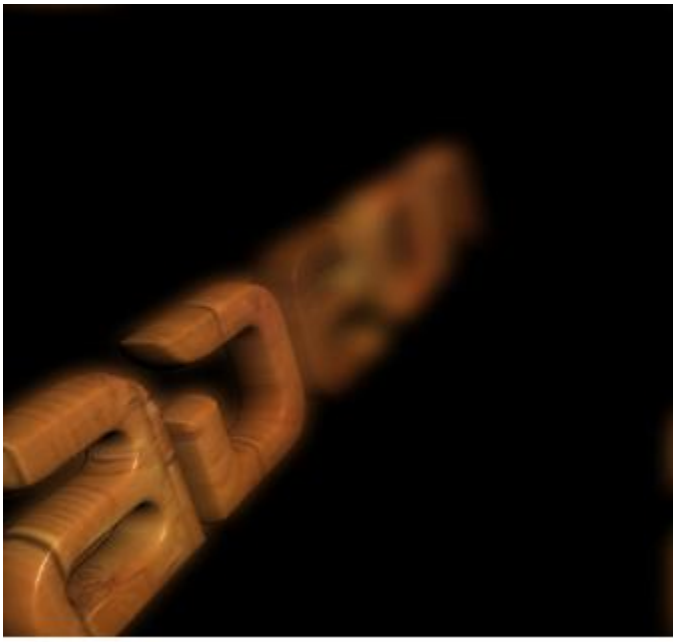

Figure 6.6: Depth of Field

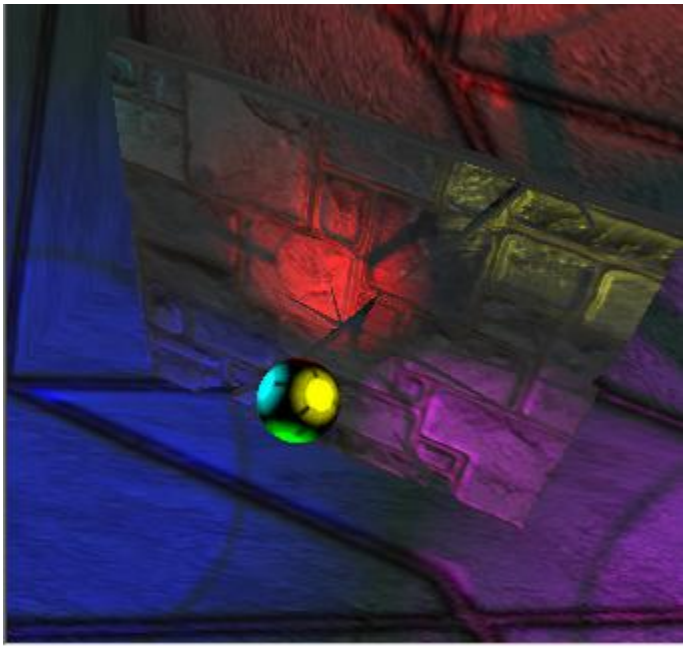

Figure 6.7: Disco Lighting

Figure 6.8: GPU Ray Tracer

the *Ambient Occlusion* emphasizes the role of texture. *Bounce* puts a stress on vertex shader and *Depth Filed* has a stress on pixel shader. The domains for which we further gathered results are described in the following:

Vertex shader The unit executes the vertex shader program on vertices. The number of ALU cycles is the lowest for *Depth Complexity*, medium for *Ambient Occlusion* and *Disco Lighting*, and substantially higher for *Bounce*. Obviously vs ALUs depend on both of the vertex shader length and the number of vertices. Considering the number of vertices for the *Clipplane* and *Bounce*, both of them have many small triangles for the objects and *Ambient Occlusion* even has more vertices than bouncing. While the length of vertex shader for *Bounce* is bigger than *Ambient Occlusion*, the total vertex shader ALU cycles for *Bounce* are bigger. However, while the number of instructions for *Disco Lighting* is bigger than *Bounce*, the total vertex shader cycles for the frame of *Disco Lighting* is still smaller (Figure 6.9). These results show that the scene

that could create a pipeline bottleneck in these units is the *Bounce* because it has small triangles (small impact on the rest of the pipeline) and it has the largest number of triangles (that are processed at the Triangle Setup) and longer instructions for the shader program. Figure 6.9 indicates the cycle counts of VS ALU for each scene. The cycle counts of the y axis is displayed as logarithm.

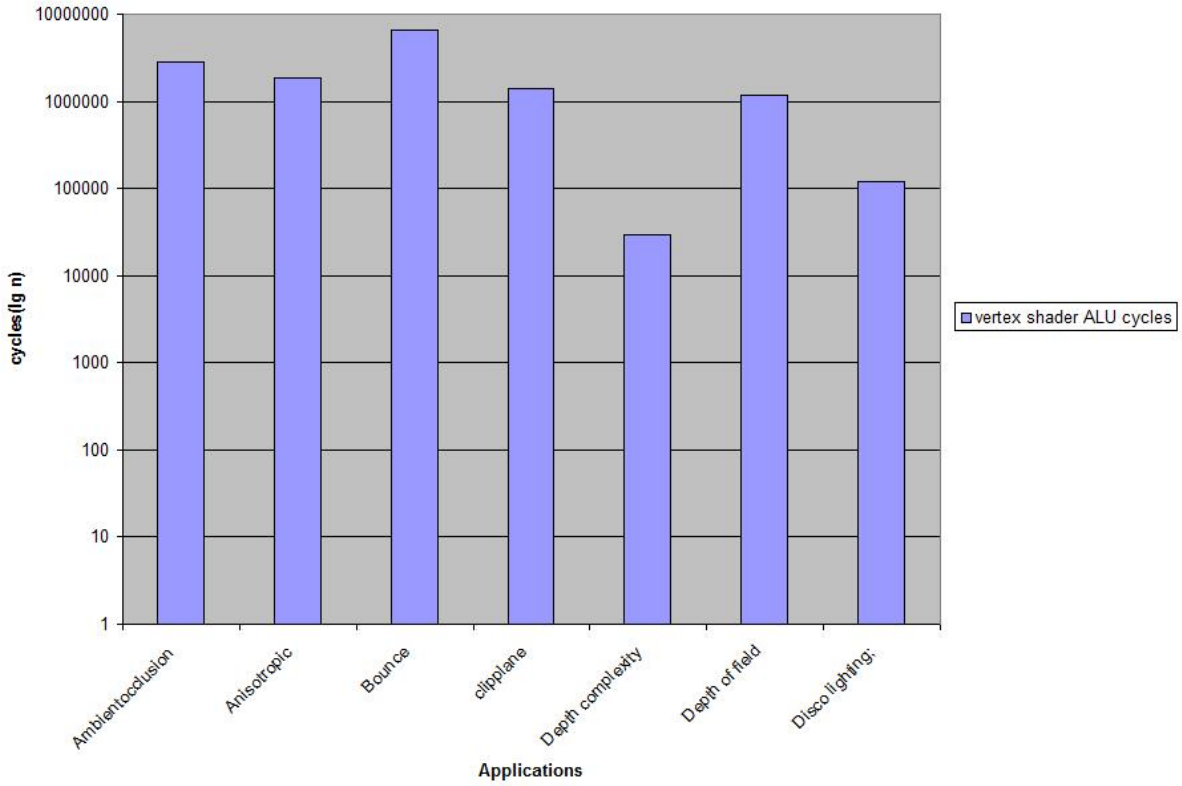

Figure 6.9: Vertex Shader Domain ALU Cycles

Rasterization The numbers of vertices and generated fragments also give an indication of the processing power required at rasterization domain. Figure 6.10 indicates that *Anisotropic* may have lightest rasterization load. Figure 6.11 shows that *Bounce* and *Clipplane* have similar stress. *Depth-of-field* and *Ambient Occlusion* may have bigger rasterization stress.

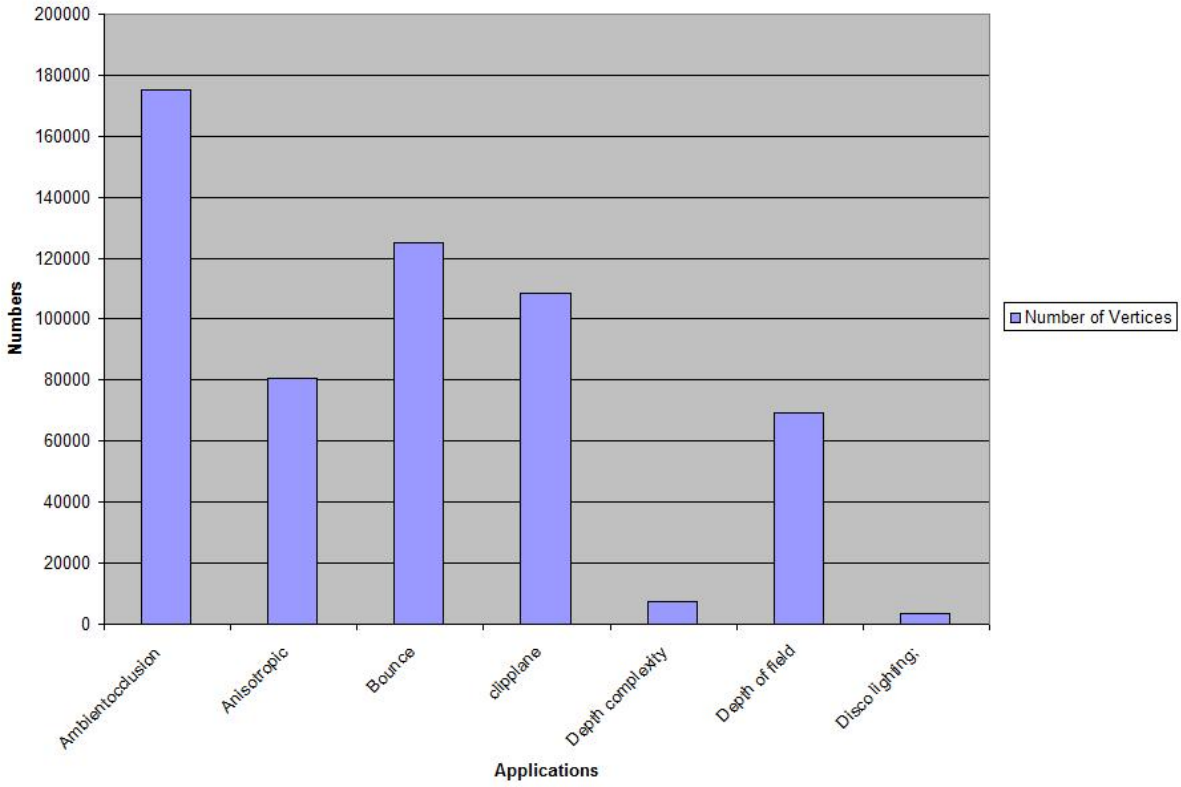

Figure 6.10: Rasterization Domain Vertices Count

- **Texture Domain** When enabled, this unit combines the color of the incoming fragment with a texture sample color. Depending on the texturing filter chosen, the texture color is obtained by a direct lookup in a texture map or by a linear interpolation between several colors (up to 8 colors in the case of trilinear interpolation). The results obtained for the texture unit are depicted in 6.12. From the Figure we can see that modern 3D graphics uses texture extensively not only for the mapping but also for the realistic lighting or shadow. Memory access operations are most expensive. Therefore, significant efforts is put into improving the efficiency for access texture. Recently, by adding a small texture cache, the traffic from the off-chip texture memory to the texture unit was substantially reduced.

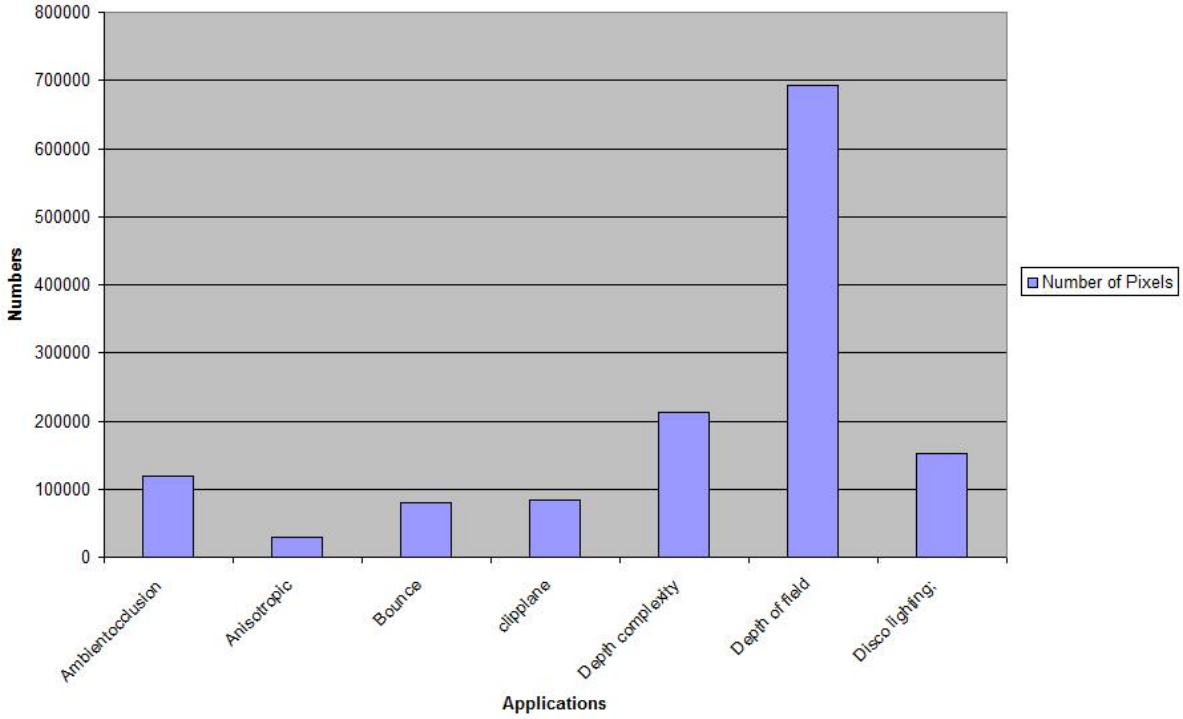

Figure 6.11: Rasterization Domain Pixel Count

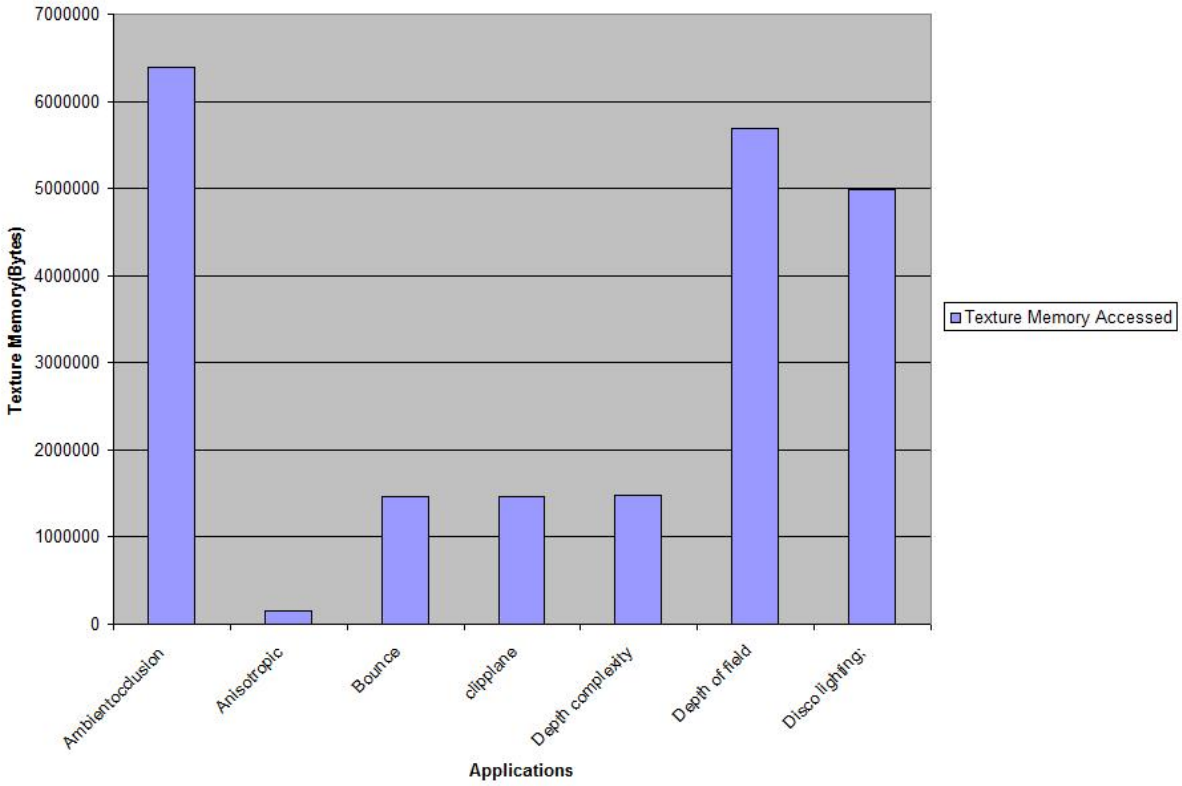

Figure 6.12: Texture Domain Memory Accessed

- **Pixel Shader.** The domain executes pixel shader program on pixels. Similarly Figure 6.13 explains that the workload in this component depends on the number of fragments and the instructions of the shader program. *Depth-of-fields* apparently has the heaviest load since it has biggest number of pixels. This Figure also indicates that this domain is not necessarily the most computationally intensive unit as we previously thought. For fixed-function pipeline, it may be true that this unit can easily become a pipeline bottleneck. However, for the programmable pipeline, sophisticated geometry models and significant visual effects can be done with vertex shader which could also make it be bottleneck.

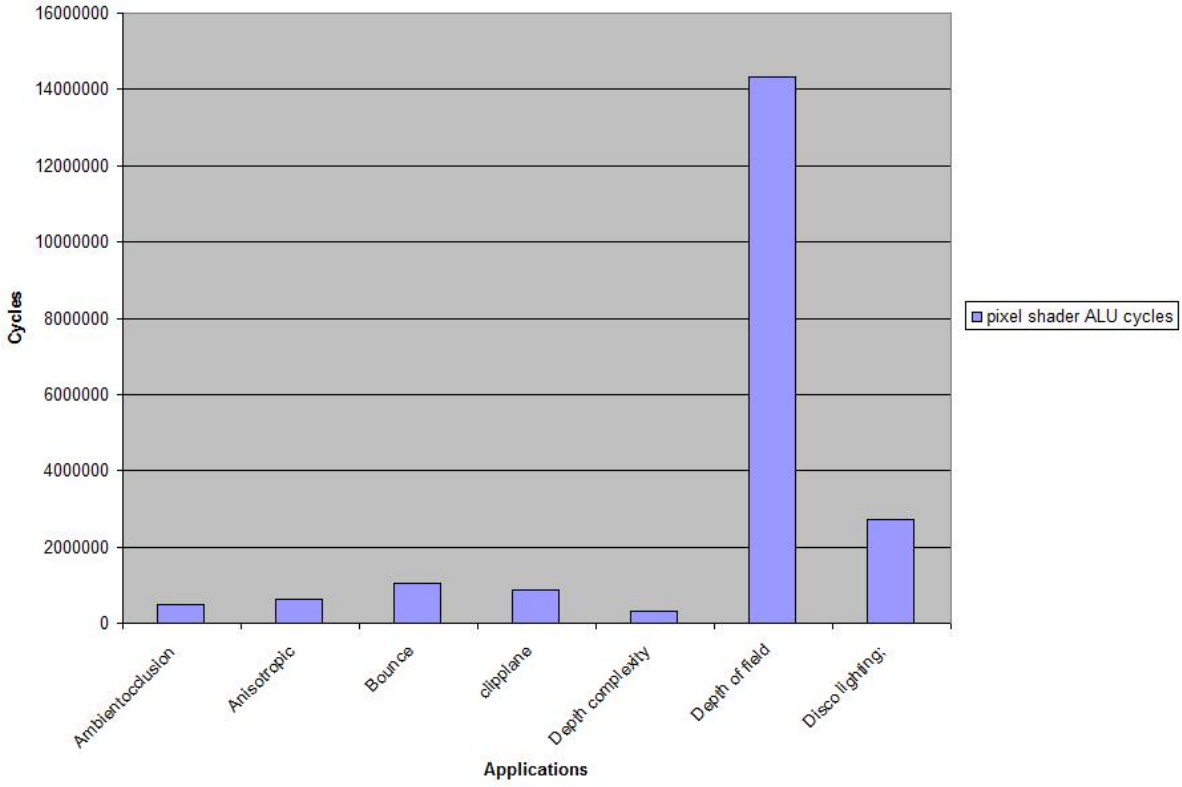

Figure 6.13: Pixel Shader Domain ALU Cycles

- **Depth buffer** This domain discards a fragment based on a comparison between its depth value and the depth value stored in the depth buffer in the fragment's corresponding position. Figure 6.14 illustrates that this unit was used intensively by all scenes. While the *Anisotropic*, *Bounce*, and *Depth Field* write all fragments that passed the depth test to the depth buffer, *Disco Lighting* writes to only 30% of the fragments that passed the test. This is expected since *Disco Lighting* uses multiple steps to apply textures to primitives and therefore it does not need to write to the depth buffer at each step. These scenes are affected by a widely used technique to render multiple passes and then blend these passes together. This suggests a method to save the bandwidth for the depth buffer, namely turn the depth buffer off for the previous passes. This unit should definitely be implemented in an aggressive manner with respect to throughput (processing power) and latency, since for instance the depth buffer read/write operations used at this unit are quite expensive.

- **Render Buffer.** Figure 6.15 shows that the Render Buffer was used intensively by all scenes. The *Clipplane* scene has the most significant workload because it has five rendering passes and uses alpha blend for four of them. This alpha blend operation combines the color of the incoming fragment with the color stored at the corresponding position in the framebuffer which includes the read operation from the framebuffer. Based on its usage and the read/write cost required, the implementation of this domain should be tuned toward performance with dedicatedly designed memory hierarchy.

In summary, the experimental results presented indicate high workload variance for the different scenes over the six domains studied.

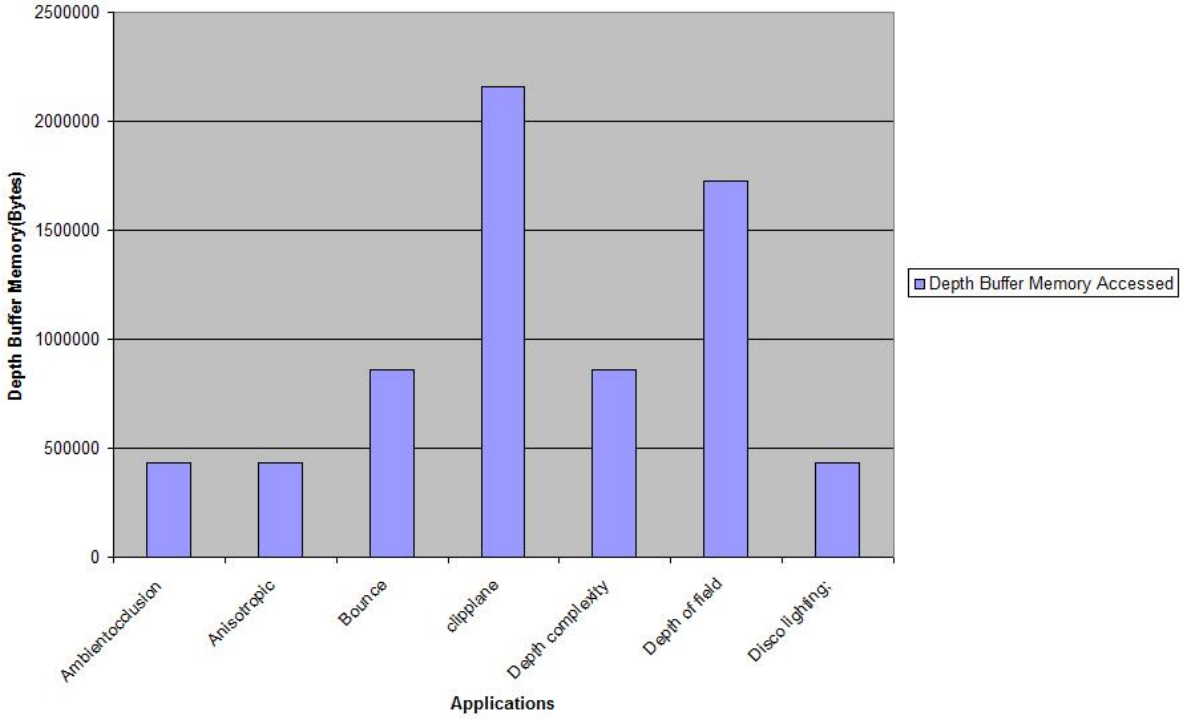

Figure 6.14: Depth Buffer Memory Accessed

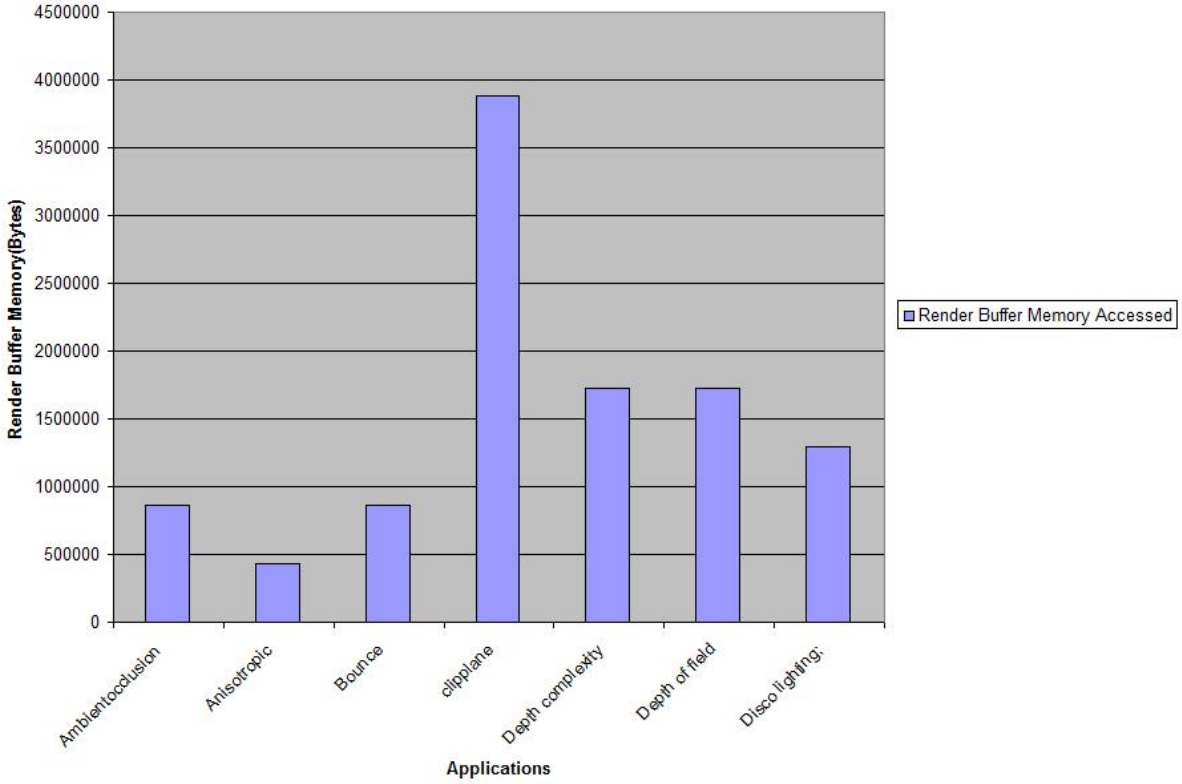

Figure 6.15: Render Buffer Memory Accessed

6.2 Energy Consumption with MCD

This section presents power and performance analysis across multiple domains for a static image which calculates the color of each pixel with the algorithms proposed by Buck[38] using pixel shaders. In addition the voltage and frequency setting of these domains is varied with increasingly rate to determine the effects of the energy save and performance. In our experiments we started with a similar frequency/voltage level.

Figure 6.16 shows the workload and performance variance for the six domains when the clock of frequency for vertex shader domain is successively reduced by half of the previous value. The y axis is the sum of idle cycle counts of multiple units for each domain while the x axis is the downclock rate with the original voltage and frequency setting (1.8v, 500MHz). This Figure indicates clearly that the pixel shader domain in the graphics pipeline is the bottleneck for this application. The idle count for the pixel shader domain is the lowest. The reason that the pixel shader has the heaviest workload is that the application needs to calculate the color for each pixel which costs around 400 instructions for completing the computation and there are many more pixels compared to the vertex. For the first three reductions of the clock frequency in vertex shader domain the performance remains the same. However, the cycle count reaches a dramatic increment at the rate of 12.5% of the full speed of the vertex shader domain. At this point, the vertex shader domain becomes the new bottleneck. It is sufficiently slow enough to block the other domains and eventually slows down the application.

A similar trend is observed from Figure 6.17. When the clock of the rasterizer domain is reduced to a certain point, the rasterizer domain becomes the new bottleneck where the tick count and idle cycle counts increase for each domain. Prior

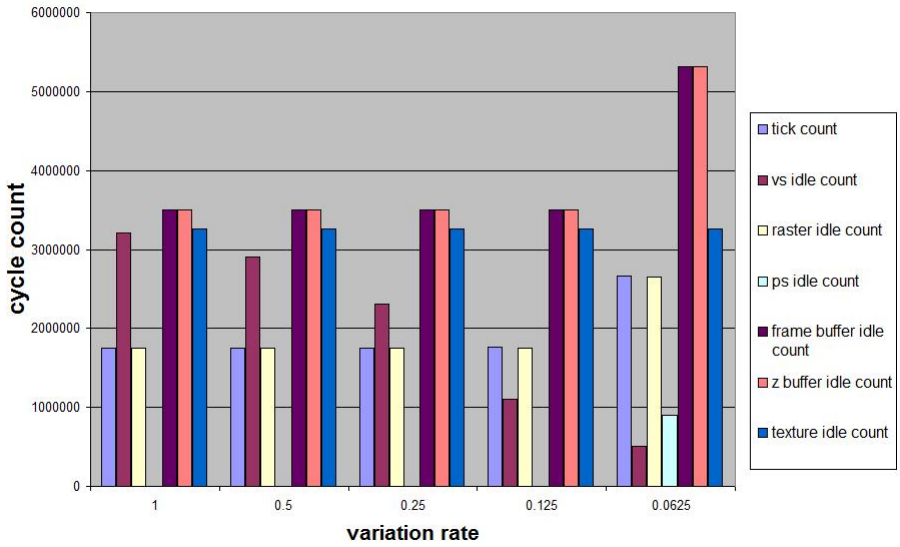

Figure 6.16: Idle Cycle Count with Variable Vertex Shader Clock Frequency

to this point, the performance changes slightly.

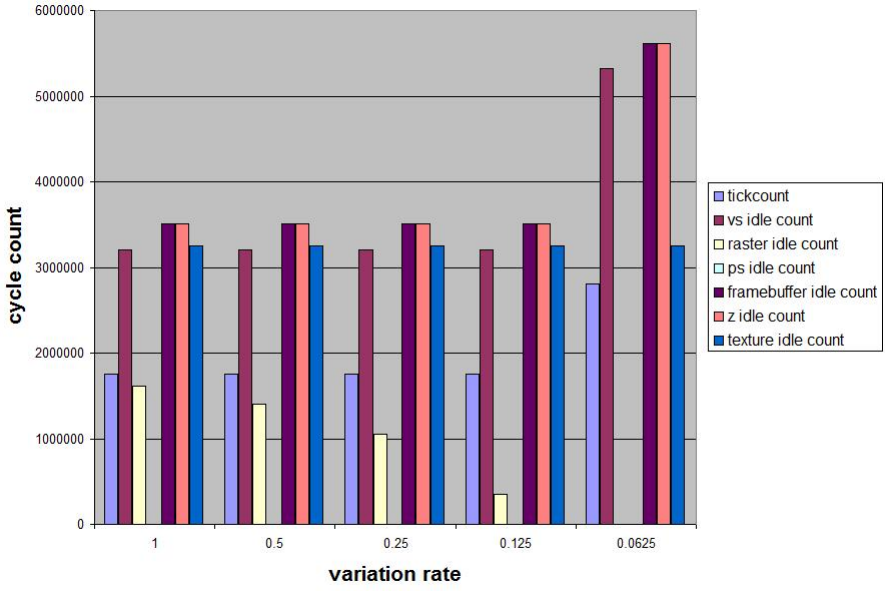

Figure 6.17: Idle Cycle Count with Variable Raster Clock Frequency

Figure 6.18 depicts the impact on the other domains when the Depth domain clock frequency is reduced by half at each step. The y axis is the cycle counts from for each domain. The x axis is the downclock rate to the original setting. The Figure illustrates that when the downclock of the domain is not the bottleneck for the application there is little effect over the entire application.

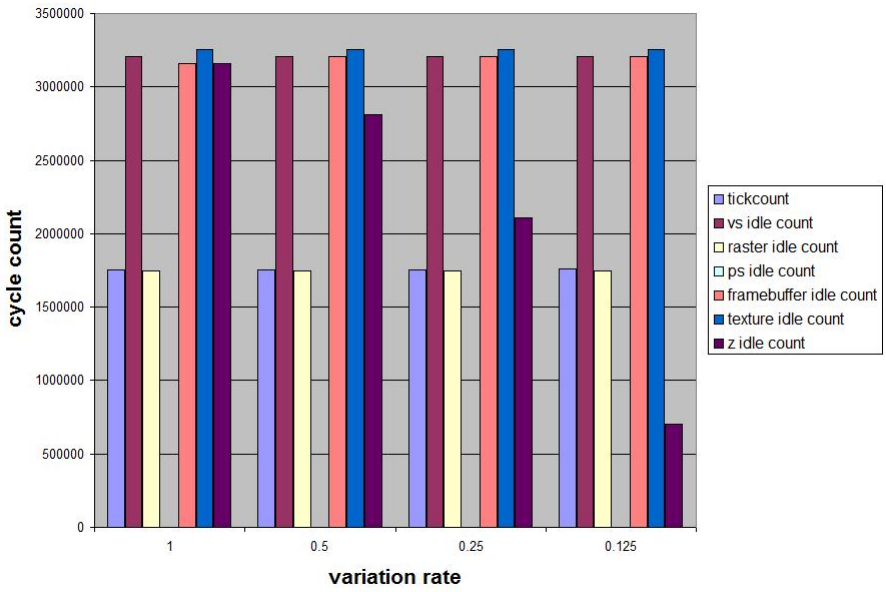

Figure 6.18: Idle Cycle Count with variable Depth Clock Frequency

However, Figure 6.19 shows a dramatically different result. When the clock for this domain is reduced, the total amount of tick counts increases significantly. Here the pixel shader is the bottleneck. When this domain slows down, it blocks the other units with subsequent performance reductions for the entire frame.

Figure 6.20 illustrates how much energy can be saved if we set different frequency/voltage levels. In this application we save most when we set the frequency of the vertex shader domain to as low as 12.5% of full speed. The column of API is to specify the frequency scale by APIs which is passed as the arguments in the tracefile. The poll column is the energy saved by simple polling the algorithm inside

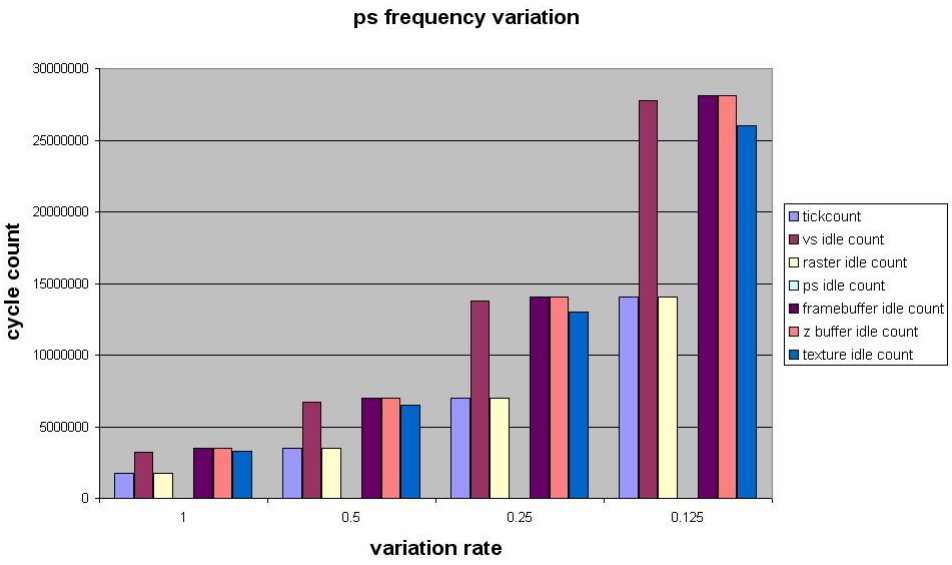

Figure 6.19: Idle Cycle Count with variable Pixel Shader Clock Frequency

the simulator.

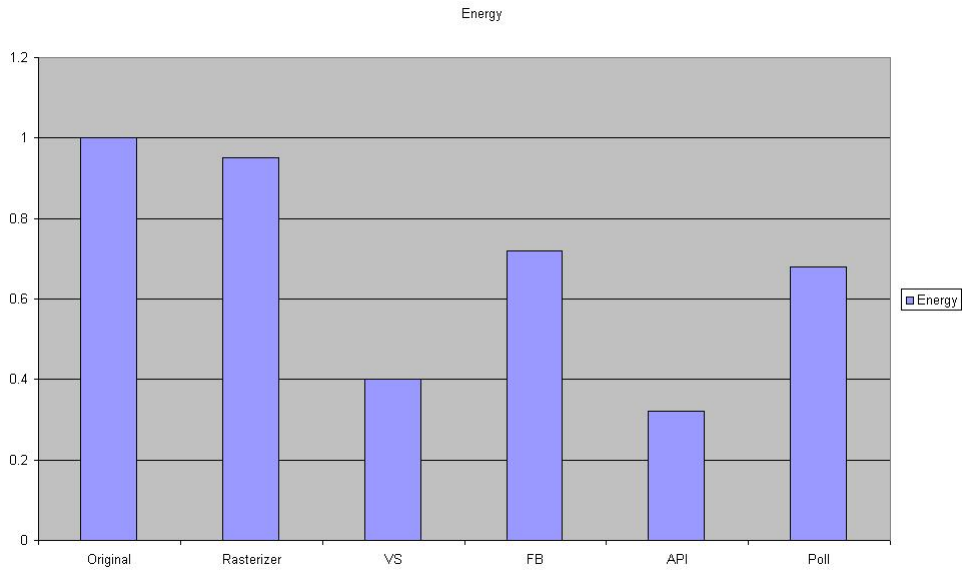

Figure 6.20: Energy Saving with the Downclock

The performance of the application is affected only slightly when we downclock

the domain which is not the bottleneck for the application. This indicates that keeping the speed as slow as possible to save energy prevents significant performance drop. On the other hand, in the domain where the most time is spent decreasing the clock it should be avoided since it slow down the application and leads to performance drop.

6.3 Workload Prediction Algorithms

In this section we examine the dynamic variability of 3D graphics workloads by modifying the applications discussed in Section 6.1 to generate consecutive frames. To simulate the varied workload in a real game scenario, those test scenes are played consecutively within a variable time interval. Figure 6.21 shows the statistic data for the frames we record.

Frames 1-28: During this segment, *Ambient Occlusion* scenes are rendered. The "status" is producing automation and rotating with the background. *Ambient occlusion* is a global lighting mode to take into account attenuation of light due to occlusion, objects blocked by others from the viewer. These frames are basically rendered with two passes, the first rendering the background texture, the next blending the environment map with the status object. In the first pass, the depth mask is off. These scenes use a lot of textures to generate the realistic environment map and lighting. Thus the texture unit in these scenes is highly loaded. The model for "status" is made up from a lot of small triangles so that the load for the vertices are higher compared with pixel parts.

Frames 29-40: During this segment a static teapot with brushed metal effect is rendered (Figure 6.2). It is one pass rendering, and only two textures of strand and gradient are used. The teapot model is made up from many tiny triangles with

a subsequent greater vertex load than pixels. The satin effect shows how rotating tangent vectors in pixel shader achieves an unconventional reflection look. The Brushed Metal effect uses gradient texture as a lookup table, and computing strand lighting generates the brushed metallic look. There are no extra writes or reads for both the depth and render buffer memories. The memory requirements for these scenes are minimum;

Frames 41-45: During this interval, a ball bounces on the ground (Figure 6.3). This shows an animation technique done by shaders. Based on time value, the ball's position is modified to make it appear squashed. The animation is calculated as instructions of vertex shaders. The geometries for the ground and ball are small meshes (see Figure 6.3) which stresses the vertex shader unit. It is rendered in two passes. The memory access (depth, render buffer, texture) is medium.

Frame 46-61: During this segment, fives planes are rendered to show the effect of clip planes (Figure 6.4). This scene shows the effect of four user-defined clip planes. The clip planes are defined by the Clip Plane Normal arrays. The XYZ components of the arrays correspond to the XYZ components of the plane's normal. The W components correspond to the distance between the plane and the origin. In these scenes the overload is close between vertex shader and pixel shader. The texture used in these scenes is similar to Frame 41-45 since these textures are used as the mapping for the planes. However, these frames require five passes for rendering. Each rendering uses alpha blend which increases the read overhead of the render buffer so that the traffic for the depth buffer and render buffer is heavy.

Frames 62-92: During the interval, the geometries are relatively simple, and no extra lighting or position calculation is used which make the vertex shader unit burden light (Figure 6.5). Small overhead on both the vertex shader and pixel

shader are expected. They use two rendering passes to show how additive blending can be used to display the "complexity", or overdraw, of a model. One pass is used to calculate the "complexity" of the model, and the second pass is used to translate the "complexity" into a more visible spectrum. Toggling the cull mode provides additional insight into the benefits of back-face culling. Extra render buffer read operations are needed.

Frames 93-96. In these scenes, six 3D characters are rendered to show a method for creating depth-of-field effects (Figure 6.6). The user is able to adjust the focal plane in the effect and change the depth that will appear in focus to the viewer. The effect is generated by interpolating between an out-of-focus texture and the in-focus texture, all based on the pixel depth. The interpolation of the texels happens on the pixel shader which leads to a heavy pixel shader burden. Additionally, the scenes normally use four passes for rendering which makes the render buffer and depth access a concern.

Frames 97-114 During this interval, a room with disco lighting is rendered (Figure 6.7). The geometries for the walls and the light are simple, big polygons requiring fewer vertices and many more pixels rendered. Thus the workload for the pixel shader is more than the vertex shader. Using a cube-map to describe the disco light, this effect shows how one may render a rotating colored light onto surrounding surfaces. The lighting for the surrounding surfaces is calculated by rotating the light-object vector by the appropriate disco rotation matrix, then looking up the resulting value from the disco light cube-map. The wall, object, and light each require a pass for rendering. The workload for the render buffer memory is quite high while the application turns the depth buffer write off for the previous two passes ultimately bringing the depth buffer memory access pretty low.

Frames 115-124. In the interval it repeats the same pattern as frames 1-28.

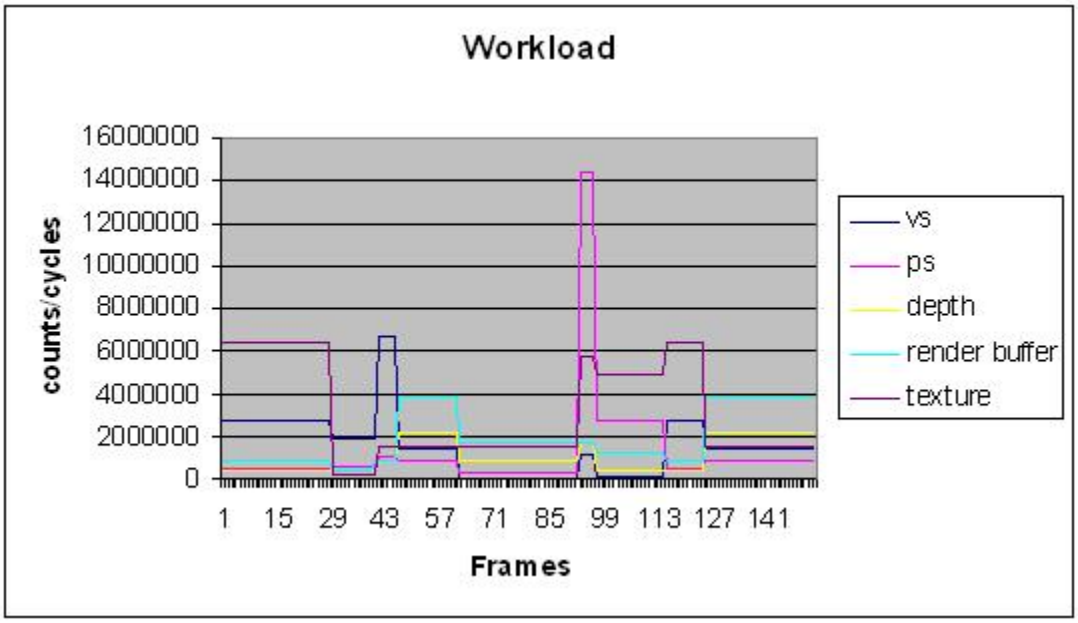

Figure 6.21: The Workload Characteristics

Frames 125-153 During the interval it repeats the same pattern as frames 46-61;

The above frame sequences illustrate a variety of factors influencing workload characteristics (Figure 6.21). It leads to two approaches to predict the workload. One is to use analytical models that compute workload requirements based on specific observable parameters such as meshes, textures, lights, and shadows. Due to the number of parameters, the challenge for this approach is the difficulty in choosing a form for the analytical model. For instance, in the game industry, each frame constitutes the following objects: brush model, alias model, texture, light map and particles. Each model has many parameters. It is computationally expensive to evaluate those models on-line, but it may be useful to relate the parameters of the models with the workloads offline. The other approach utilizes information which can be easily extracted from the graphics pipeline. Our signature based workload prediction algorithms is based on this approach.

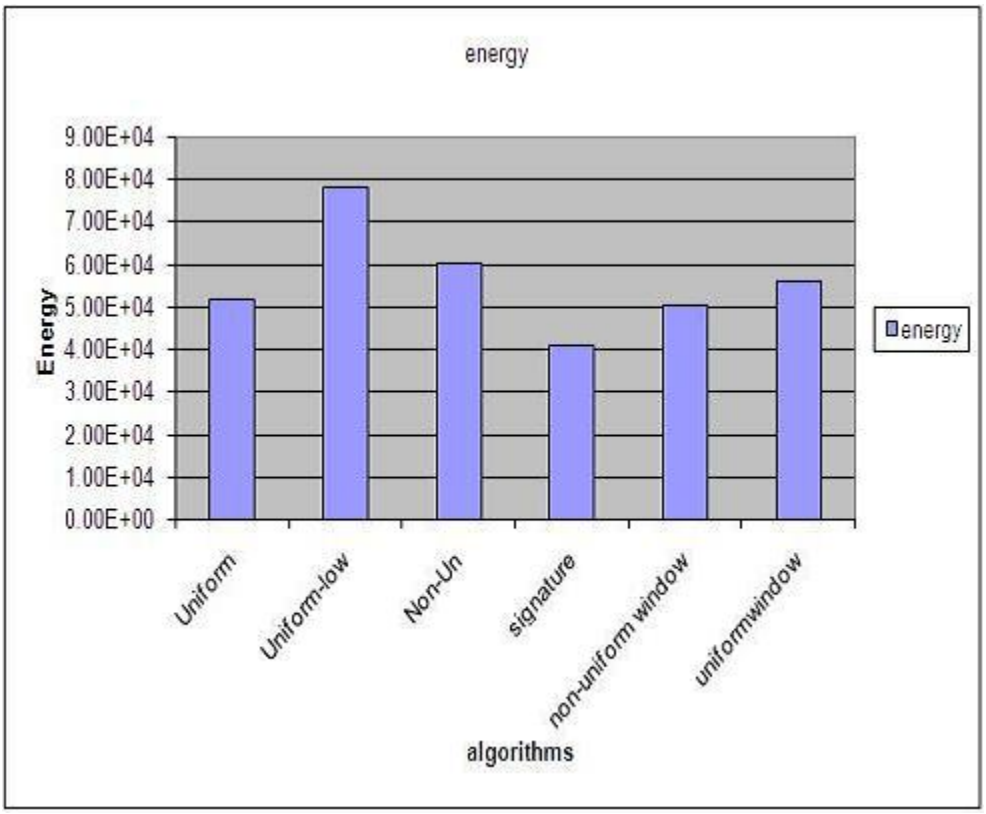

Figure 6.22: Energy Consumption Comparison

Figure 6.21 shows the workload variance along the frames. The x axis is the frame number, and the y axis for the numbers of collected statistic data. Those five series are vertex shader cycle counts, pixel shader cycle counts, depth buffer accessed counts, render buffer accessed counts and texture memory accessed counts. The workload in each domain varies along the frames.

Figure 6.22 and Figure 6.23 show energy consumption and discharge rate comparison for the DVFS algorithms.

The following is an explanation of the analytic algorithms tested:

(1) **Uniform:** The voltage and frequency of each domain are the same and set to the maximum of the stages.

(2) **Uniform Low:** The voltage and frequency of each domain are the same and

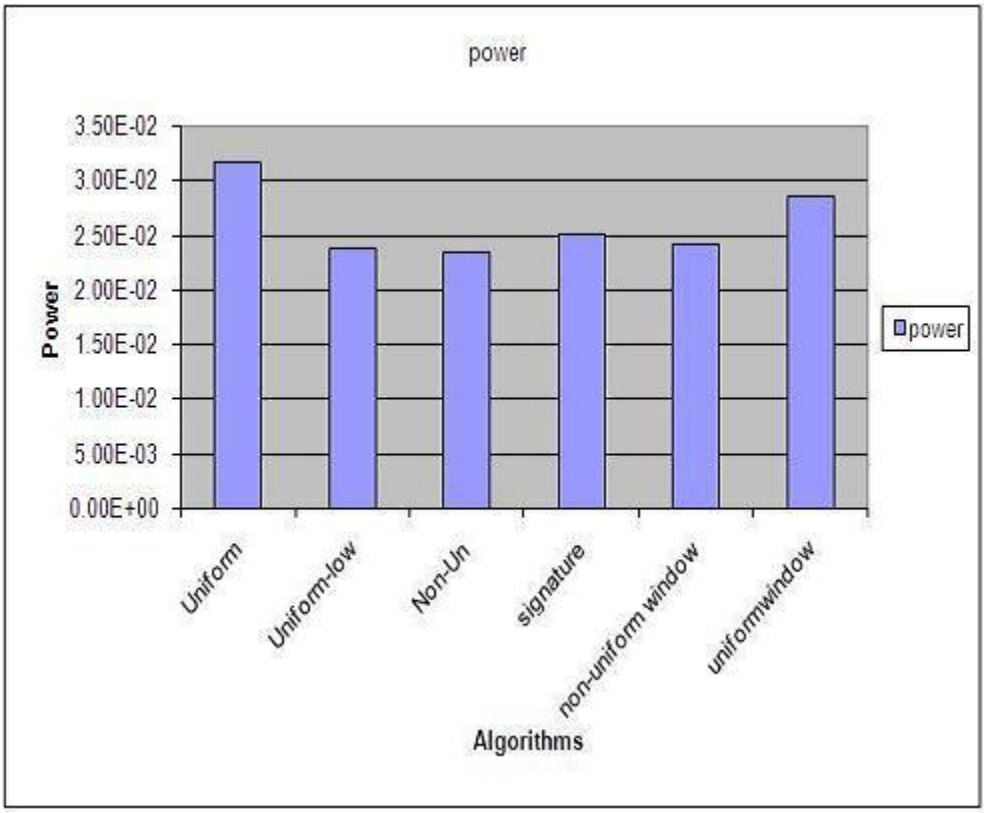

Figure 6.23: Power Comparison

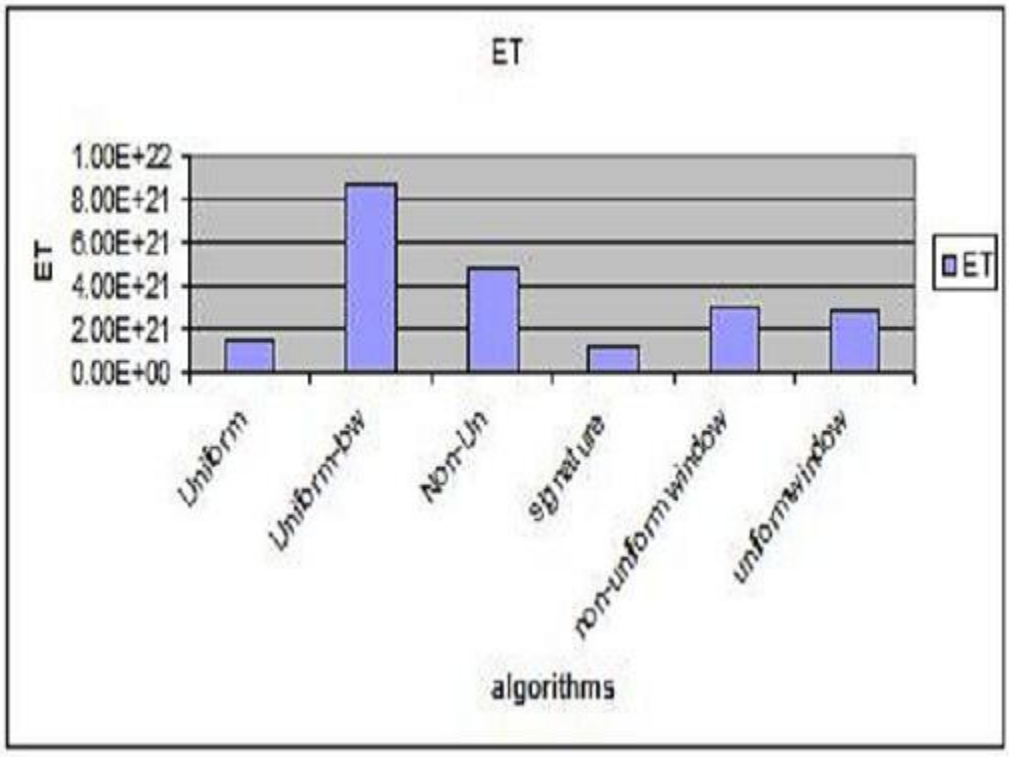

Figure 6.24: The Energy Delay Product

set to the minimum of the stages.

(3) **Non Uniform:** The voltage and frequency of each domain are different and keep the same for the entire application.

(4) **Signature:** The voltage and frequency of each domain are different and varied on the signature-based workload prediction algorithm.

(5) **Non-uniform Window:** The voltage and frequency of each domain are different and varied on the AVGN algorithm.

(6) **Uniform Window:** The voltage and frequency of each domain are the same, but varied on the AVGN algorithm.

The x axis on the three figures (Figure 6.22 to Figure 6.24) provide the algorithms. The y axis of Figure 6.22 is the energy consumed by each algorithm. The y axis of Figure 6.23 is the energy consumed by each algorithm. The y axis of Figure 6.24 is the energy and time product by each algorithm. The first two algorithms demonstrate how well voltage scaling works and how useful non-uniform voltage scaling is. The energy consumed by the uniform-low is the greatest in Figure 6.22 .

Interestingly, the uniform high scheme is actually quite efficient. In Figure 6.24, the product of Energy and Time for this scheme is second lowest only to the signature based algorithm. However, from Figure 6.23, the power used by uniform is the highest. In this scheme, the voltage and frequency setting is maximum. The discharge rate (measured by power) also turns out to be the maximum. The entire application could be finished in a shorter time and overall energy consumed reduced. The uniform low scheme has the lowest discharge rate with the greatest energy consumption and requires more time to finish the task. Also, the static non-uniform scheme does not necessarily save energy though the discharge rate for it may be lower. This makes sense because of the workload variations in each domain through the frames. Even though the setting for each domain is optimal for the first frames,

the following workload could change dramatically. If the setting cannot change with the workload, it may have negative effects on the following frames, increase time executing the task, and consume more energy.

It is therefore better to use dynamic scaling policies with on-line 3D applications for uncertain workloads. We can see signature, non-uniform window, and uniform window all have better energy efficiency than the static policies. The static non-uniform algorithm has better results with static scenes than an application which has little workload variation among each domain in the pipeline. Good examples of this are some of the general-purpose application on GPUs. The pattern for the resource usage is generally fixed and does not change along applications.

Comparing non-uniform window with uniform window, both power and energy consumption for non-uniform are better than uniform. This matches results seen in the previous section. Non-uniform can make better use of the resources for each domain and reduce the idle cycles for the units not bottlenecked in the pipeline. The non-uniform window can save 13% energy over the uniform window scheme.

Signature is the optimal algorithm. It saves approximately 20% energy compared to the uniform high scheme and almost 50% compared with the uniform lowest one. It is better than the AVGN based algorithm since it is not limited to the history of frames nearby. In 3D games, the frames usually have good continuity in workload so the scenes typically look smooth and brusque changes are not expected. However, in some situations the frames can change dramatically. For instance, in air plane fighting games the player can change the view from full to overlook. The overlook view might produce the entire scene from a perspective requiring fewer details, and the geometries behind the scenes are simple with fewer polygons. While in full view mode the objects inside the plane, in other planes, and objects nearby should be rendered in high detail, and this usually means many more polygons and levels of

detail. In our case, the last frames (115-153) actually repeat the frames rendered in the beginning. The bottleneck is in the vertex shader unit for frames 1-61, while after Frames 62 the bottleneck moves to the pixel shader, and from frame 115 the bottleneck returns to the vertex shader. An AVGN based algorithm cannot work well for these changes even when using a weight to evaluate the previous frames in the window size.

In conclusion, the signature based workload prediction algorithm for DVFS is better for applications with a widely varying workload. Additionally, dynamic scaling is generally better than static schemes with a varied workload. Finally, if the voltage or frequency setting can not be scaled dynamically, it is better to run applications with dynamic workload at full speed than minimum speed.

Chapter 7

Related Work

Hardware-based voltage and frequency scaling is a widely used technique for low power circuit design[6]. It has been studied for general purpose processors and real-time systems. There are several DVFS approaches that make use of the asynchrony of memory access to the CPU clock during task execution. In [16] and [17], DVFS techniques are proposed in which frequency is lowered in memory-bound region of a program with little performance drop. DVFS techniques fall into three categories: hardware-based, OS-based, and application-directed methods. Hardware-based [10] methods measures system utilization in hardware and choose a system-wide speed setting based on the current utilization. OS-based approaches determine a system-wide CPU setting based on the processor demands of the active tasks [9, 23]. Application-directed approaches exports the entire burden of power management to the user level [22].

There are algorithms proposed for applications with predictable computational workloads such as audio [5]. Other digital signal processing intensive applications [4, 34] describe a DVFS technique for MPEG decoding to reduce the energy consumption while maintaining a quality of service. With graphics processor units

becoming more powerful and programmable, and with the increased popularity in handheld devices, many researchers show interests in applying DVFS to GPUs. Unlike the DVFS applied to general purpose processors and real-time systems, DVFS applied to GPUs can be for the single processor instead of a system. In [29], the GPU is divided into triple dynamic voltage and frequency scaling power domains to minimize the power consumption at a given performance level. From the architecture view of a GPU, [19] divides the GPU into three domains. The power of three different power domains is managed by continuous co-locking of voltage and clock, dynamically varying clock frequency and supply voltage level from 90 MHz to 200 MHz and from 1.0 V to 1.8 V, respectively.

Workload prediction has been studied for general purpose processors for a long time[43][36]. [8] propose working set signatures on microarchitectures with configurable units like caches and CPU resources. The research similar to the thesis in [24] addresses the problem of workload prediction for mobile 3D graphics and monitors specific parameters of the 3D pipeline providing workload prediction. However, this research focuses on workload prediction with parameters which are complicate and have significant cost to generate and compare the signatures. For the power analysis in 3D graphics pipeline, [25] provides a quantitative analysis of the power consumption of 3D graphics pipeline. However, this analysis are limited to fixed-function pipeline and some of the parameters used for discuss are out-of-date. Our experimental framework is conceptually similar to the one described in [18].

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis presented a detailed quantitative analysis of the workload variations and imbalances of different domains of a mobile 3D graphics pipeline, and the DVFS based power savings that exploit such variations and imbalances. Six possible clock domains for dynamic voltage and frequency scaling were proposed and shown to be useful for saving energy. We propose power-friendly APIs to control the power saving policies. Additionally we propose a signature-based algorithm to predict the future workload and minimize the energy consumption without significant frame rate drop. We also compared the energy saving by dynamic scaling algorithms with static MCD algorithm, signature-base algorithms with AVGN algorithms. Our studies show that signature-based DVFS strategies achieve success for examples with widely varied workload, with savings of over 50% for best case.

8.2 Future work

There remains future work within the algorithms to decide the scaling point and scaling factor for the DVFS algorithms. This thesis only uses linear decrease or increase of a fixed delta value. Furthermore, the sample interval for passing through the prediction algorithm is another interesting point for future consideration. We need to investigate the balance between the DVFS algorithm overhead and performance. Another interesting investigation is to determine workload incurred in rendering for each object offline and to determine the number of occurrences of each object online.

Bibliography

[1] AMD Corporation, AMD Rendermonkey, http://ati.amd.com/developer/rendermonkey/.

[2] AMD Corporation, AMD GPU Tools, http://developer.amd.com/GPU/.

[3] L. Benini and G. Micheli, System-level Power Optimization: Techniques and Tools. ACM Transactions on Design Automation of Electronic Systems, pp. 115-122, April 2000.

[4] T. Burd, T. Pering, A. Stratakos, and R. Brodersen, A Dynamic Voltage Scaled Microprocessor System, IEEE Journal of Solid- State Circuit, vol. 35, no.11, pp. 1571-1580, November 2000.

[5] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos, Data Driven Signal Processing: An Approach or Energy Efficient Computing, ISLPED-96: ACM/IEEE International Symposium on Low Power Electronics and Design, pp.347-352, 1996

[6] A. Chandrakasan, S. Sheng, and R. Brodersen, Low Power CMOS Digital Design. IEEE Journal of Solid-State Circuits, vol.27, no.4, pp. 473-484, April 1992.

[7] C. Chiasserini and R. Rao, Pulsed Battery Discharge in Communication Devices, In The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, pp. 88-95, August 1999.

[8] A. S. Dhodapkar and J. E. Smith, ManagingMulti-Configuration Hardware via Dynamic Working Set Analysis, in Proc. Int. Symp. Computer Architecture, pp. 233-244, May 2002

[9] K.Flautner and T.Mudge, Vertigo:Automatic performance-setting for Linux, In Proc. of OSDI'02, pp. 251-266, December 1995.

[10] M.Fleischmann, Longrun Power Management-Dynamic Power Management for Crusoe Processors, Technical report, Transmeta Corporation, 2001.

[11] R. Gonzalez and M. Horowitz, Energy Dissipation in General Purpose Microprocessors, IEEE Journal of Solid-State Circuits, vol.31, no.9 pp. 277-284, September 1996.

[12] D. Grunwald, P. Levis, Polices for Dynamic Clock Scheduling, Process of 4th conference on Operating System Design and Implementation, vol. 4, 2000.

[13] G. Humphreys,M.Houston,R.Ng,S.Ahern,R.Frank, Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations, ACM Transactions on Graphics,vol. 21,no.3, pp. 693-702, July 2002.

[14] Intel Corporation, Mobile Pentium iii Processor in Bga2 and Micro-pga2 Packages, Datasheet Order 245302-002, 2000.

[15] Khronos Group, OpenGL ES Overview, Http://www.khronos.org/opengles,2005.

[16] C. Hsu and U. Kremer, Compiler-directed Dynamic Voltage Scaling for Memory-bound Applications, Technical Report DCS-TR-498, Department of Computer Science, Rutgers University, August 2002.

[17] C. Hsu and U. Kremer, Single Region vs. Multiple regions: A Comparison of Different Compiler-directed Dynamic Voltage Scheduling Approaches,Proc. of Workshop on Power-Aware Computer Systems, February 2002.

[18] G. Lafruit, L. Nachtergaele, K. Denolf, and J. Bormans, Power-constrained Design for Multimedia Table of Contents,3D Computational Graceful SESSION, Ses. 35 , pp. 592 - 597, 2006.

[19] J. Lee, B. Nam and H. Yoo, Dynamic Voltage and Frequency Scaling (DVFS) Scheme for Multi-Domains Power Management, Solid-State Circuits Conference, 2007. ASSCC apos;07. IEEE Asian, pp. 360 - 363, November,2007.

[20] D. Linden. Author, Handbook of Batteries, *McGraw-Hill*, vol. 17, pp. 1-100, 1987.

[21] D. Linden (editor), Handbook of Batteries, 2nd ed. McGraw-Hill, New York, 1995.

[22] X. Liu and P. Sherwy, Chameleon:Application Level Power Management with Performance Isolation, International Multimedia Conf. Proceedings of the 13th annual ACM International Conf. on Multimedia, pp. 839-848, 2005.

[23] J.R.Lorch and A.J.Smith, Improving Dynamic Voltage Scaling Algorithms with PACE, In Proc. of ACM Sigmetrics'01, pp. 50-61,June 2001.

[24] B. Mochocki,K. Lahiri,S. Cadambi ,X. Sharon Hu, Signature-based Workload Estimation for Mobile 3D Graphics, Annual ACM IEEE Design Automation Conference archive Proceedings of the 43rd annual conference on Design automation table of contents San Francisco, CA, USA,2007.

[25] B. Mochocki, K. Lahiri, and S. Cadambi, Power Analysis of Mobile 3D Graphics, in Proc. Design Automation Test Europe (DATE) Conf., pp. 502-507, March 2006.

[26] V. Moya, C. Gonzalez, Shader Performance analysis on a Modern GPU Architecture. Micro 38,2005.

[27] V. Moya, C. Gonzalez, A Single(Unified) Shader GPU Microarchitecture for Embedded Systems. Hi-PEAC, 2005.

[28] V. Moya, C. Gonzalez, A Cycle-level Execution-Driven Simulator for Modern GPU Architectures. Micro 38,2005.

[29] B. G. Nam, J. Lee, K. Kim, S.J. Lee, and H.-J. Yoo, A Low-Power Handheld GPU using Logarithmic Arithmetic and Triple DVFS Power Domains. In Proceedings of ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware, pp. 73-80, 2007.

[30] Alexey Nicolaychuk, RivaTuner Tools, http://www.guru3d.com/index.php?page=rivatuner.

[31] NVIDIA Corporation (2007), NVIDIA CUDA Compute Unified Device Architecture Programming Guide, http://developer.download.nvidia.com.

[32] Nvidia Corporation, http://www.nvidia.com/page/8800_tech_briefs.html.

[33] J. Paradiso and T. Starner, Energy Scavenging for Mobile and Wireless Electronics, IEEE Pervasive Computing vol. 4, no. 1, pp. 18-27, January 2005.

[34] M. Pedram, W. Cheng, K. Dantu, K. Choi, Frame-based Dynamic Voltage and Frequency Scaling for a MPEG decoder, iccad,2002 International Conference on Computer-Aided Design (ICCAD '02), pp732-737, 2002

[35] T. Pering, T. Burd, and R. Brodersen, The Simulation of Dynamic Voltage Scaling Algorithms, In IEEE Symposium on Low Power Electronics. IEEE Symposium on Low Power Electronics, 1995.

[36] T. Pering, T. Burd, and R. Brodersen, The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms, in Proc. Int. Symp. Low Power ElectronicDesign, pp. 76-81, Aug. 1998.

[37] T. Pering, T. Burd, and R. Brodersen, Voltage scheduling in the Parm Microprocessor System, In Proceedings of the 2000 International Symposium on Low Power Design, August 2000.

[38] J. Purcell , I. Buck, R. William , and P. Hanrahan, P. Ray Tracing on Programmable Graphics Hardware, in Proc. SIGGRAPH 2002, pp. 703 - 712, 2002

[39] J. W. Sheaffer and D. P. Luebke and K. Skadron, A Flexible Simulation Framework for Graphics Architectures, in Proc. of the Eurographics Conference on Graphics Hardware, pp. 85-94. Aug. 2004

[40] J. W. Sheaffer and D. P. Luebke and K. Skadron, Studying Thermal Management for Graphics-Processor Architectures, IEEE International Symposium on Performance Analysis of Systems and Software, 2005

[41] D. Trebilco, GLIntercept, http://glintercept.nutty.org/.

[42] M. Weiser, B. Welch, A. Demers, and S. Shenker, Scheduling for Reduced CPU Energy, In First Symposium on Operating Systems Design and Implementation, pp.13-23, November 1994.

[43] F. Yao, A. Demers, and S. Shenker, A Scheduling Model for Reduced CPU Energy, in Proc. Symp. Foundations of Computer Science, pp. 374-382, Octomber. 1995.

[44] V. Zyuban, Unified Architecture Level Energy-efficiency Metric, In Proceedings of the 2002 Great Lakes VLSI Symposium, pp. 24-29,2002.